Системы реального времени

Системы обработки данных - СОД

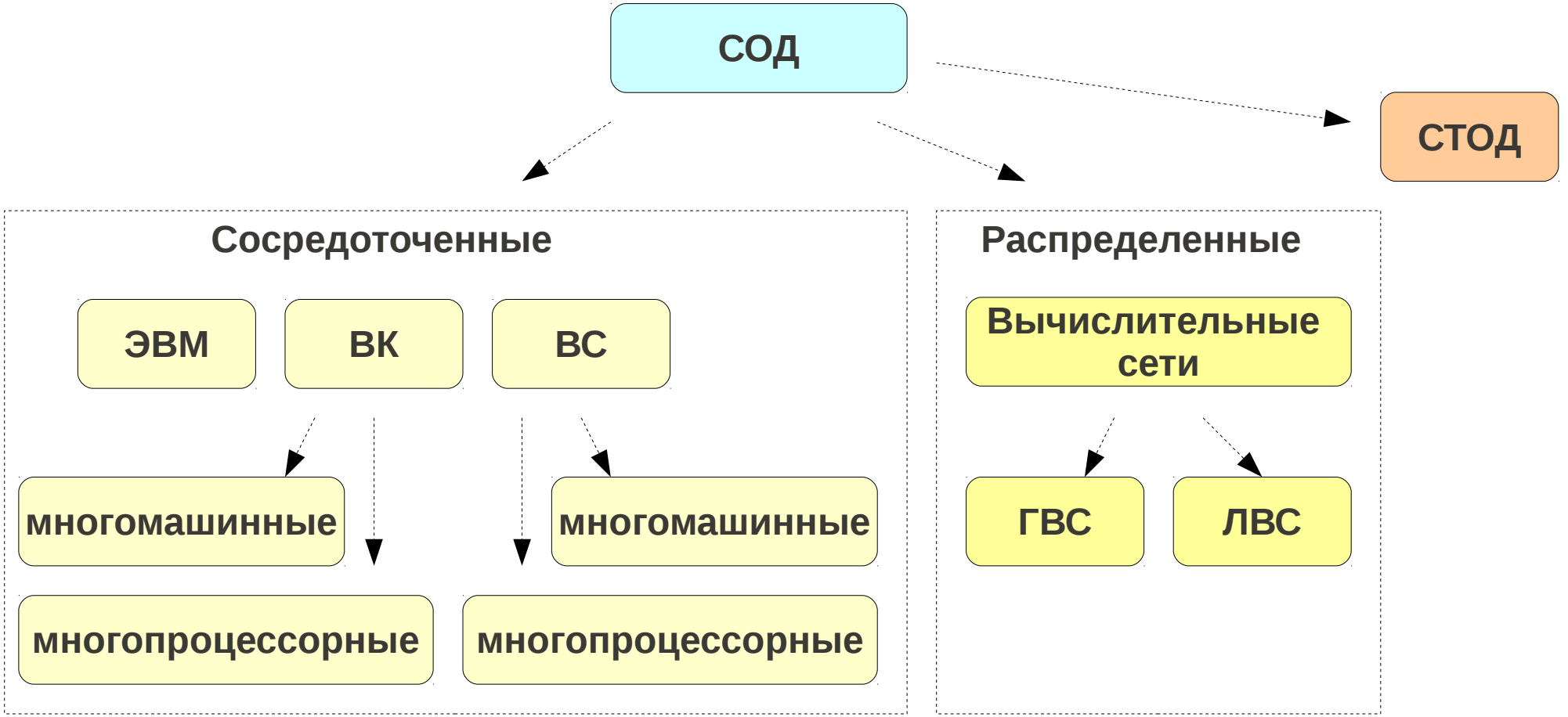
СОД – это совокупность технических средств и программного обеспечения, предназначенных для информационного обслуживания пользователя и технических объектов.

СОД

SOFTWARE

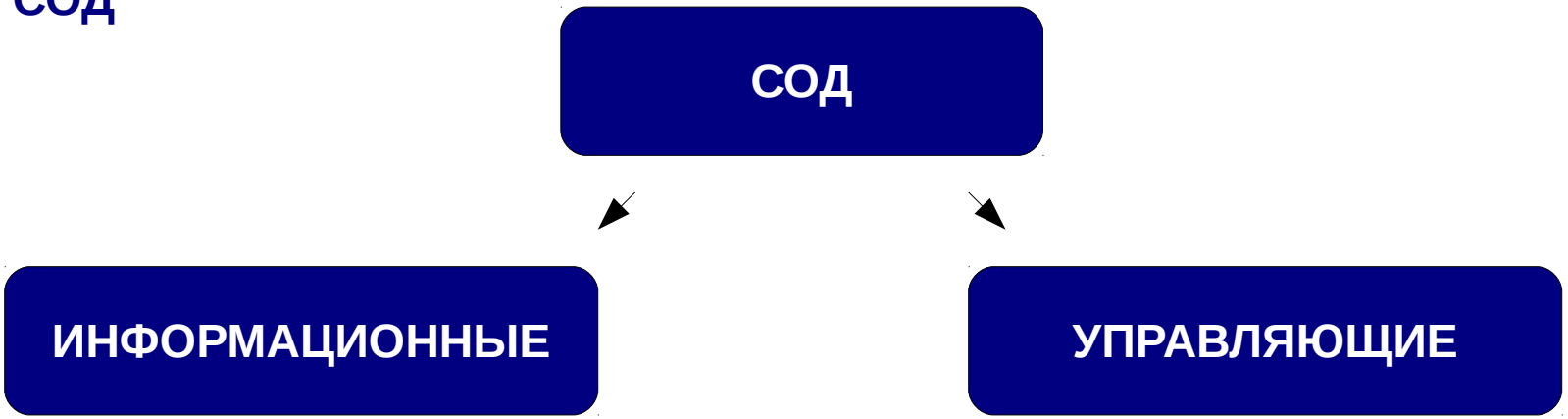
HARDWARE

Классификация СОД



ВК — вычислительный комплекс
ВС — вычислительная система
СТОД — системы телеобработки данных
ГВС — глобальные вычислительные сети

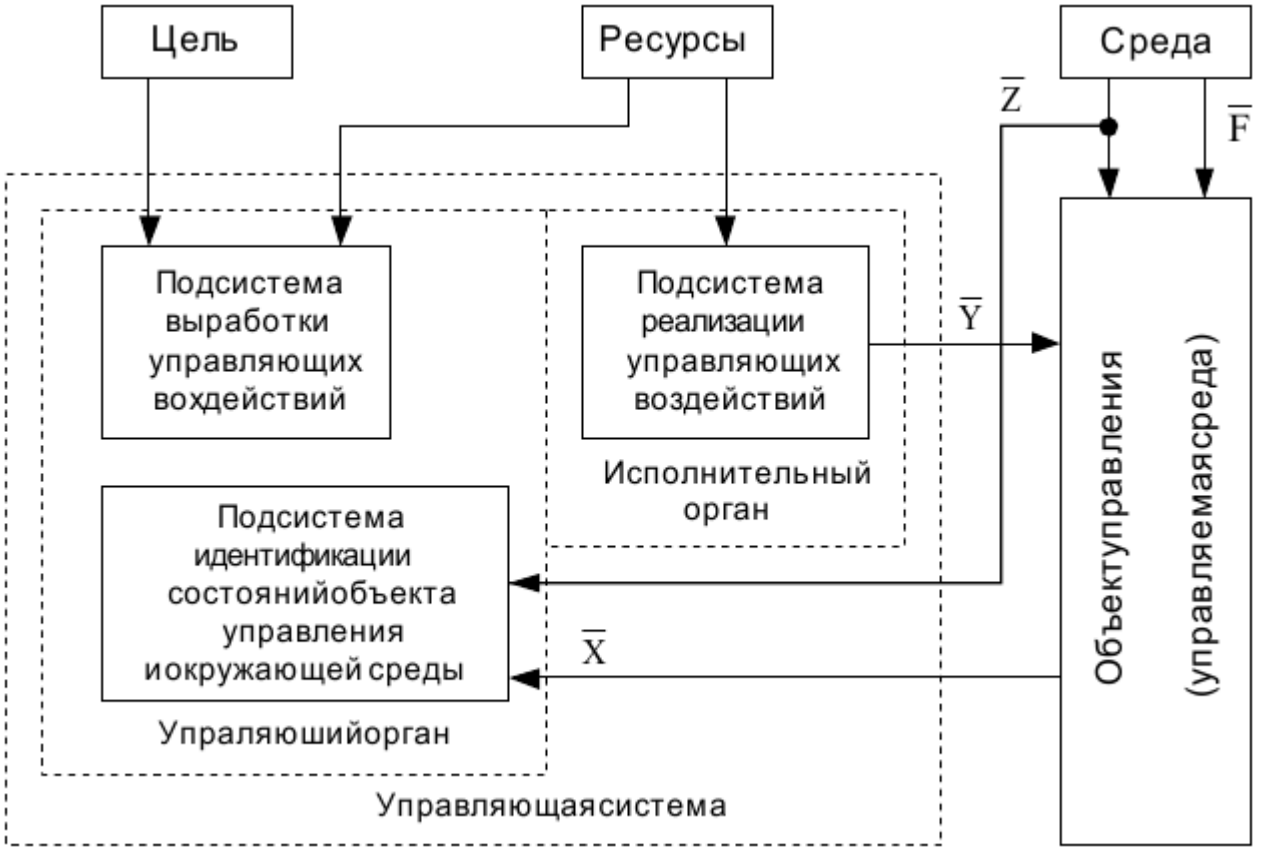
Типы СОД



Назначение ***информационных*** систем – **поиск и анализ информации, потребителем которой является человек**. Основу алгоритмов работы такой системы составляют программы логической обработки данных. Допустимое время обработки данных определяется максимально возможным временем ожидания. В таких системах, как правило, объем входной информации невелик, но в них присутствует большие постоянные и медленно изменяющиеся массивы данных.

Назначение ***управляющих*** систем – **целенаправленное изменение состояние объекта или управление процессом функционирования объекта**. Для управления необходимо знать состояние объекта в заданный момент времени, каковы внешние воздействия на объект, т.е. воздействие окружающей среды, какова цель управления и какие средства воздействия на объект имеются в распоряжение системы управления.

Связи между объектами управления и системой управления



$Y=(y_1,y_2,\ldots,y_n)$ — управляющие воздействия

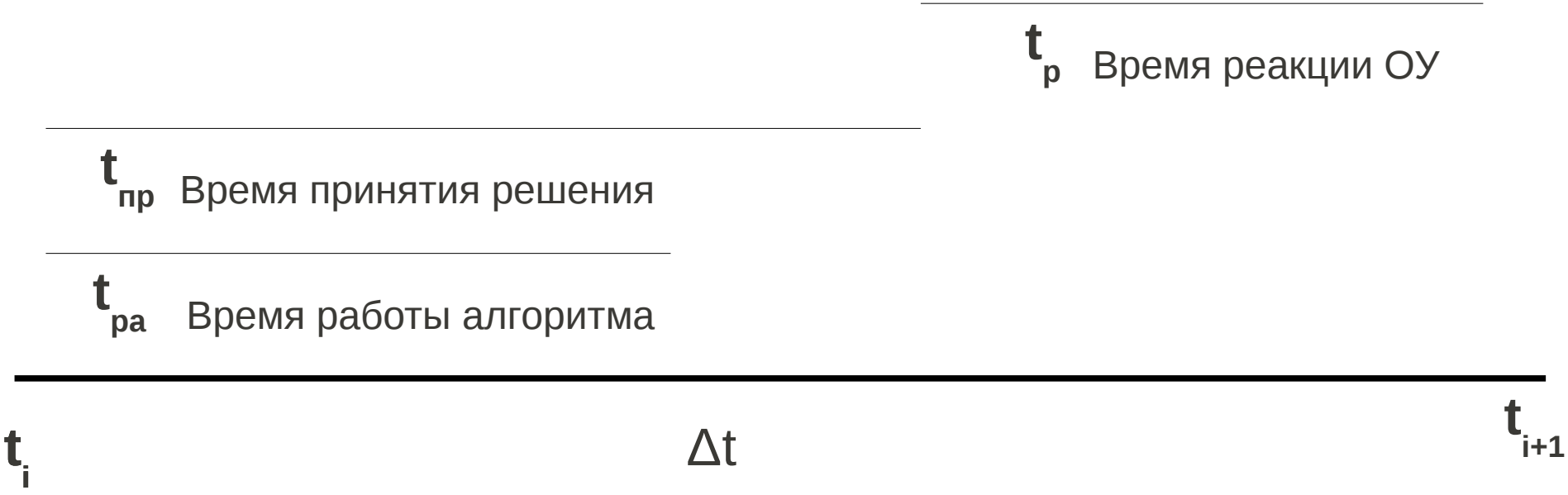
$Z=(z_1,z_2,\ldots,z_n)$ - контролируемые возмущения

$F=(f_1,f_2,\ldots,f_n)$ - неконтролируемые возмущения

$X=(x_1,x_2,\ldots,x_n)$ – контролируемые величины

Параметры объекта управления, существенные для системы управления

Увеличение количества параметров приводит к усложнению системы управления



$t_{пр} + t_p \le t - t_i$, $t<t_{i+1}$ $\quad$ $t_{пр} + t_p \le t - t_i = \Delta t$, $t<t_{i+1}$

$t_{ра} \le t_{пр} = \Delta t - t_p$

Понятие масштаба реального времени

Основным параметром, определяющим скорости всех процессов в системе управления, является время переходного процесса в объекте управления (t_p). Оно определяет, какой масштаб времени следует выбрать при анализе и синтезе системы.

Например, для системы управления самолетом такой единицей временной шкалы является доля секунды, для системы управления предприятием – часы сутки.

Особенности работы СОД в масштабе реального времени:

чрезвычайно малое время, отведенное для принятия решения, соизмеримое со временем переходного процесса в объекте

сложность алгоритмов решения задач управления и практически мгновенное использование результатов решения для управления

недопустимость, как преждевременной выдачи управляющих сигналов, так и их запаздывания

Время в СОД

При работе системы в РВ используется либо астрономическое, либо относительное время.

Системы, работающие в астрономическом времени, несколько сложнее, систем, работающих с относительным временем. Это связано, с необходимостью периодически сверять астрономическое время с источником точного времени. Учет астрономического времени нужен при решении задач навигации, слежения за планетами и космическими объектами.

Относительное время широко используется в системах, которые управляют технологическим процессом и техническими системами. Это время отсчитывается либо с момента включения системы, либо с начала какой, либо фазы технологического процесса.

Системой реального времени **называют аппаратно-программный комплекс, реагирующий в предсказуемые моменты времени на не предсказуемый поток внешних событий.**

система должна успевать отреагировать на события, произошедшие на объекте, в течение времени, критического для этого события. Величина критического времени для каждого события определяется объектом и самим событием, и может быть разной, но время реакции системы должно быть предсказано при создании системы; отсутствие реакции в предсказанное время считается ошибкой для СРВ;

система должна успевать реагировать на одновременно происходящие события; даже если два или больше внешних событий происходит одновременно, система должна успеть среагировать на каждое из них в течении интервалов времени, критических для этих событий.

Классификация СРВ

Системы жесткого реального времени СЖРВ – это подкласс СРВ, в котором неспособность получения правильных результатов за определенный крайний срок может закончиться катастрофическим отказом системы, т.е. СЖРВ не допускают никаких задержек реакции системы ни при каких условиях так как:
- результаты могут, оказаться бесполезны в случае их опоздания;
- может произойти катастрофа в случае задержки реакции;
- стоимость опоздания может оказаться бесконечно великой.

Примерами СЖРВ являются все системы аварийной защиты, бортовые системы управления, системы управления атомными станциями и железнодорожным транспортом.

Системы мягкого реального времени СМРВ – это те СРВ, в которых способность реагировать в краткие сроки действительно требуется, но отказ выполнения не приводит к отказу системы.

СМРВ характеризуется тем, что задержка реакции не критична, хотя и может привести к увеличению стоимости результатов и уменьшению производительности системы в целом.

Основные отличия между системами жесткого и мягкого РВ: СЖРВ **никогда** не опоздает с реакцией на событие, СМРВ **не должна** опаздывать с реакцией на событие.

Характеристики СОД

При проектировании СОД стремятся обеспечить наиболее полное соответствие системы своему назначению. Степень соответствия системы своему назначению называют *эффективностью* или *качеством системы.* Для сложных систем, к которым относят и СОД, не удается определить эффективность одной величиной, поэтому ее представляют набором величин, называемых *характеристиками системы.*

Набор характеристик формируется таким образом, чтобы в совокупности они давали наиболее полное представление об эффективности системы.

Производительность – это характеристика вычислительной мощности системы, определяющая количество вычислительной работы, выполняемой системой за единицу времени.

Производительность технических средств определяется их **быстродействием** - числом операций, выполняемых устройством за единицу времени, например, быстродействие процессора $1*10^6$ операций в секунду, быстродействие внешнего ЗУ двести обращений в секунду, скорость ввода/вывода канала 6000 символов в секунду.

Оценка производительности быстродействием устройства называется *номинальной производительностью.* Номинальная производительность ($V_{ном}$) характеризует потенциальные возможности устройств, которые не могут быть использованы полностью.

Чтобы оценить влияние структуры системы на быстродействие устройств используется специальная характеристика, называемая ***комплексной производительностью***. Комплексная производительность ($V_{ком}$) оценивается набором быстродействий в составе комплекса технических средств.

Один из подходов к оценке комплексной производительности состоит в следующем. Некоторым образом определяется типовая смесь операций ввода, обращения к внешней памяти, обработки и выводы данных, на основе которых создается синтетическая искусственная программа, порождающая процесс с заданной смесью операций. Путем прогона такой синтетической программы и измерения времени ее выполнения оценивается комплексная производительность системы (очевидно, что $V_{комп} < V_{ном}$) .

Показателем использования устройства в процессе работы системы является **загрузка**. *Загрузка* i-го устройства

$$\rho_i = \frac{T_i}{T}$$

T_i – время, в течении которого устройство работало, T – продолжительность работы системы ($\rho_{i \le 1}$).

Если в системе N устройств и загрузка каждого ровна соответственно ρ_1, ρ_2, ρ_3, …ρ_n, то количество работы, выполняемой устройствами с быстродействием $v_1, v_2, v_3, \ldots, v_n$, равно соответственно ($\rho_1 v_1$, $\rho_2 v_2$, $\rho_3 v_3$, … ,$\rho_n v_n$). Совокупность этих значений ($\rho_1 v_1$, $\rho_2 v_2$, $\rho_3 v_3$, … ,$\rho_n v_n$) характеризует производительность технических устройств с учетом простоев, возникающих в процессе функционирования системы.

Таким образом, оценка фактической производительности системы сводится к оценке загрузки устройств в конкретных условиях работы.

Для СОД, находящихся в эксплуатации, или разрабатываемых для конкретного применения, класс решаемых задач определен полностью, т.е. определена рабочая нагрузка. В таком случае производительность оценивается на рабочей нагрузке и называется ***системной производительностью*** или просто *производительностью.*

В этом случае производительность оценивается числом задач, решаемой системой за единицу времени.

Эта оценка информативна только для конкретной области СОД и не о чем не говорит, если не определен класс задач. По этой причине системная производительность используется тогда, когда анализируются различные варианты организации одной СОД, и не может применяться для сравнения СОД, работающих с различными наборами задач.

Время ответа – **это длительность промежутка времени от момента поступления запроса на выполнения задания в систему до момента вывода из системы результатов выполнения данного задания.**

В общем случае, время ответа величина случайная, что обусловлено следующими факторами:

- влиянием исходных данных на число операций ввода, обработки и вывода данных и не предсказуемостью значений исходных данных;
- влиянием состава смеси задач, одновременно находящихся в системе, и непредсказуемостью состава смеси из-за случайности момента поступления задач на обработку в систему;

Время ответа как случайная величина наиболее полно характеризуется функцией распределения (интегральная характеристика) F(u<T) или функции плотности вероятности (дифференциальная характеристика) f(u). Чаще всего время ответа оценивается средним значением, которое определяется как статистическое среднее случайной величины U_i, i=1,n

$$U = \frac{1}{n}\sum_{i=1}^{n} u_i$$

В общем случае **время ответа** складывается из двух составляющих:
времени выполнения задачи
времени ожидания.

Время выполнения задачи при отсутствии параллельных процессов равно суммарной длительности всех этапов процесса обработки ввода, обращения к внешней памяти, процессорной обработки и вывода.

Время ожидания - это сумма промежутков времени, в течении которых задача находилась в состоянии ожидания требуемых ресурсов. Ожидания возникают при мультипрограммной обработке, когда ресурс, необходимый данной задаче занят другой задачей и первая задача не выполняется, ожидая освобождения ресурса.

Режимы обработки данных

Режим обработки данных это способ выполнения заданий, характеризующийся порядком распределения ресурсов системы между заданиями.

Режим обработки данных обеспечивается управляющими программами операционной системы, которые выделяют заданиям оперативную и внешнюю память, устройства ввода-вывода, процессорное время и прочие ресурсы в соответствующем порядке с учетом атрибутов заданий: имен пользователей, приоритетов заданий, сложность задач.

Режим обработки данных порождает соответствующий режим функционирования системы, который проявляется в порядке инициирования задач, в предоставлении одним задачам преимущественное право на использование ресурсов, в организации ввода данных, хранение программ и данных в оперативной памяти, вывода данных.

Порядок распределение ресурсов между заданиями влияет на время пребывания задания в системе и на производительность системы.

Классификация режимов обработки данных

По числу дновременно обрабатываемых заданий

ОДНОЗАДАЧНЫЕ (однопрограммные)

МНОГОЗАДАЧНЫЕ (мультипрограммные)

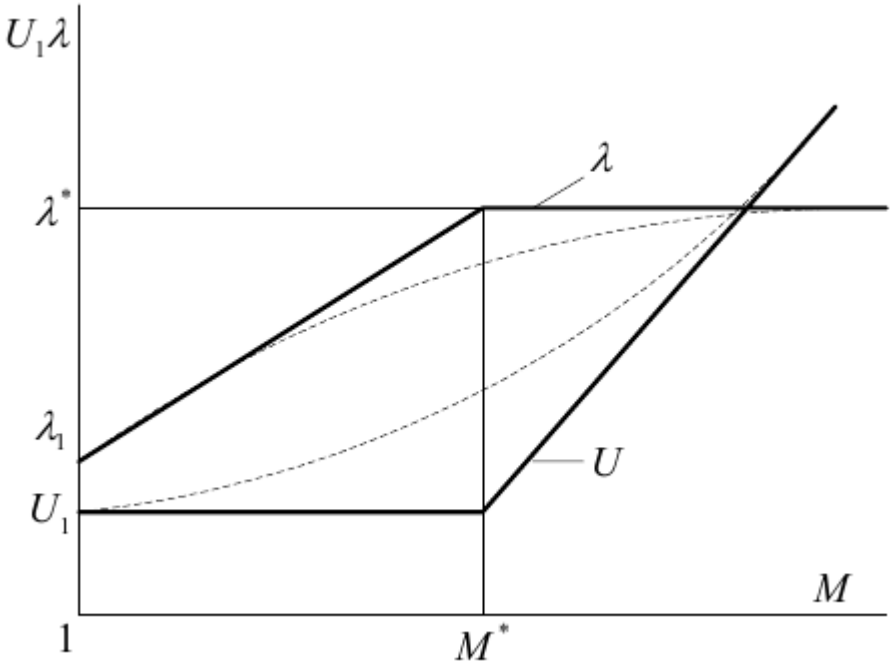
В общем случае процесс решения задачи сводится к последовательности этапов процессорной обработки, ввода, вывода данных и обращений к внешним ЗУ.

При этом задача в каждый момент времени обрабатывается каким-либо одним устройством, а остальные устройства не могут использоваться до завершения работы этого устройства, и, следовательно, могут распределяться для выполнения других задач.

Режим обработки, при котором в системе одновременно обрабатывается несколько задач, называется ***мультипрограммной обработкой***.

При этом процессы обработки, относящиеся к разным задачам, одновременно выполняются разными устройствами системы, способными функционировать параллельно. В этом случае говорят, что СОД функционирует в мультипрограммном режиме.

Цель мультипрограммирования – увеличение производительности системы. Число задач, одновременно находящихся в системе, называется *уровнем мультипрограммирования*.



Зависимость времени ответа и производительности системы от уровня мультипрограммирования

В однопрограммном режиме интенсивность выходного потока λ_1, среднее время ответа U1=Uвых

С увеличением уровня мультипрограммирования увеличивается вероятность того, что большее число устройств одновременно заняты выполнением заданий, но вместе с тем вероятность того, что несколько задач одновременно обращаются к одному устройству, достаточно мала, и время ожидания оказывается незначительным.

Однако при $M=M^*$ возникает ситуация, когда, по крайней мере, одно из устройств оказывается полностью загруженным. Дальнейшее увеличения уровня мультипрограммирования не приводит к увеличению производительности, при этом начинает резко возрастать время ответа, т.к. все большее число заданий ожидают момента освобождения устройств.

Значение M^* называют **точкой насыщения мультипрограммной смеси**. Если задачи преимущественно используют одно устройство, то значение M^* невелико и может быть равным единице.

Если задачи загружают все устройства, то значение M^* определяется числом устройств в системе.

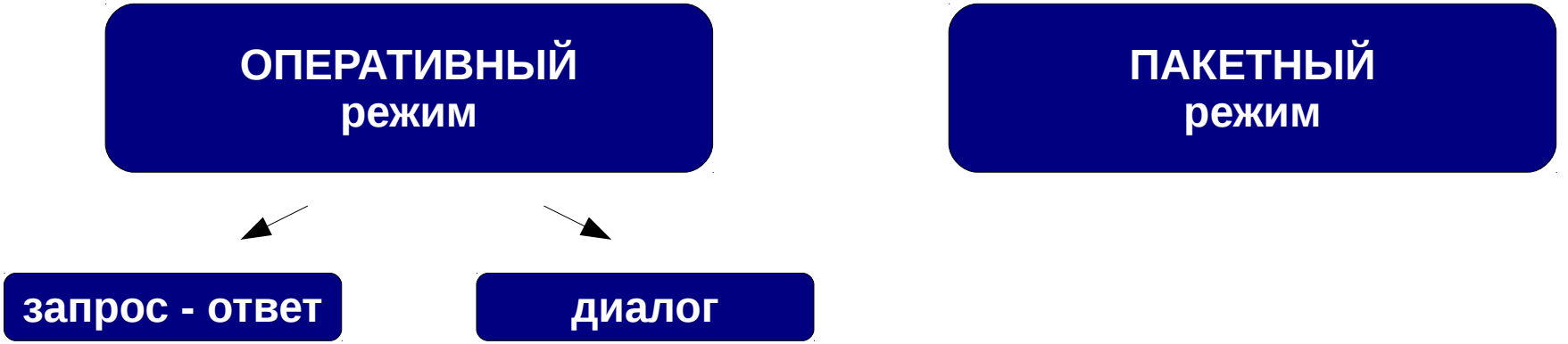
Если в системе N устройств, загрузка которых ровна ρ_1, ρ_2, ρ_3, … ,ρ_n, то среднее число задач, которые находятся на обработке в системе, равно суммарной загрузке всех устройств:

$$m = \sum_{i=1}^{N} \rho_i$$

Остальные (M-m) задач находятся в состоянии ожидания. Число одновременно выполняемых задач m называется ***коэффициентом мультипрограммирования***. Оно равно отношению производительности системы в мультипрограммном режиме к ее производительности в однопрограммном режиме: $m=\lambda/\lambda_1$.

Таким образом, коэффициент мультипрограммирования является показателем увеличения производительности системы за счет мультипрограммирования

С точки зрения режима взаимодействия пользователя с системой различают два режима обработки:



Применительно к СОД, предназначенным для информационного обслуживания пользователей (а не технических объектов и систем) оперативная обработка данных характеризуется:

- малым объемом вводимых и выводимых данных и вычислений, приходящихся на одно взаимодействие пользователя с системой;

- высокой интенсивностью взаимодействия и вытекающим отсюда требованием уменьшения времени ответа.

Оперативная обработка используется в банковских справочных системах, и системах резервирования билетов.

В рамках оперативной обработки выделяют два подрежима: **запрос–ответ и диалоговый.**

Пакетная обработка данных характеризуется:

- большим объемом вводимых и выводимых данных и вычислений, приходящихся на взаимодействие пользователя с системой;

- низкой интенсивностью взаимодействия и допустимостью большого времени ответа.

Пакетная обработка данных применяется в вычислительных центрах научно-технического профиля, в системах обработки учетно-статистических данных.

Режим реального времени

Применительно к техническим объектам и техническим системам имеет место **обработка в реальном масштабе времени**.

В этом режиме каждая задача инициируется либо периодически, либо при возникновении определенных ситуаций в системе. При этом темп инициирования задач и время получения результатов решений жестко регламентируются динамическими свойствами управляемого объекта. Это означает, что на время решения задач управления налагаются ограничения, определяющие предельно допустимое время ответа.

Режим, при котором организация обработки данных подчиняется режиму процессов вне СОД, называется *обработкой в реальном масштабе времени.*

Обработка в реальном масштабе времени обеспечивается за счет:

- выбора структуры СОД и быстродействия устройств в соответствии с задачами обработки и требованиями к времени ответа;

- способов организации процессов обработки, обеспечивающих требуемое время ответа при ограниченной производительности устройств и заданной структуре СОД.

Режим телеобработки данных это режим обработки данных при взаимодействии пользователей с СОД через каналы связи.

Наличие процесса передачи данных в этих системах налагает ограничения на форму и время обмена данными между пользователями и СОД. Эти ограничения приводят к необходимости специальных способов организации данных и доступа к ним, что в свою очередь отражается на структуре прикладных программ, используемых в режиме телеобработки.

Режим телеобработки характеризуется спецификой доступа пользователей к системе обработки и системы обработки к данным пользователя. При этом пользователи могут работать в режимах **пакетном, диалоговом или запрос-ответ**.

Требования к операционным системам реального времени

ОСРВ должна быть многонитевой (многопоточной) и прерываемой.

ОСРВ должна быть предсказуемой, что означает максимальное время выполнения того или иного действия, которое должно быть известно заранее и должно соответствовать требованиям приложения.

Первое требование состоит в том, что ОС должна быть многонитевой по принципу абсолютного приоритета (прерываемой). Планировщик должен иметь возможность прервать любую нить и предоставить ресурс той нити, которой он более необходим. ОС (и аппаратура) должны также обеспечивать прерывания на уровне обработки прерываний.

ОСРВ должна обладать понятием приоритета для потоков.

Проблема в том, чтобы определить, какой задаче требуется ресурс. В идеальной ситуации ОСРВ отдает ресурс нити или драйверу с ближайшим крайним сроком (так называемые ОС, управляемые временным ограничением (deadline driven OS)). Чтобы реализовать это, ОС должна знать время, требуемое каждой из выполняющихся нитей для завершения (до сих пор не существует ОС, построенной по этому принципу, так как он слишком сложен для реализации), поэтому разработчики ОС принимают иную точку зрения: вводится понятие уровня приоритета задачи, и временные ограничения сводят к приоритетам.

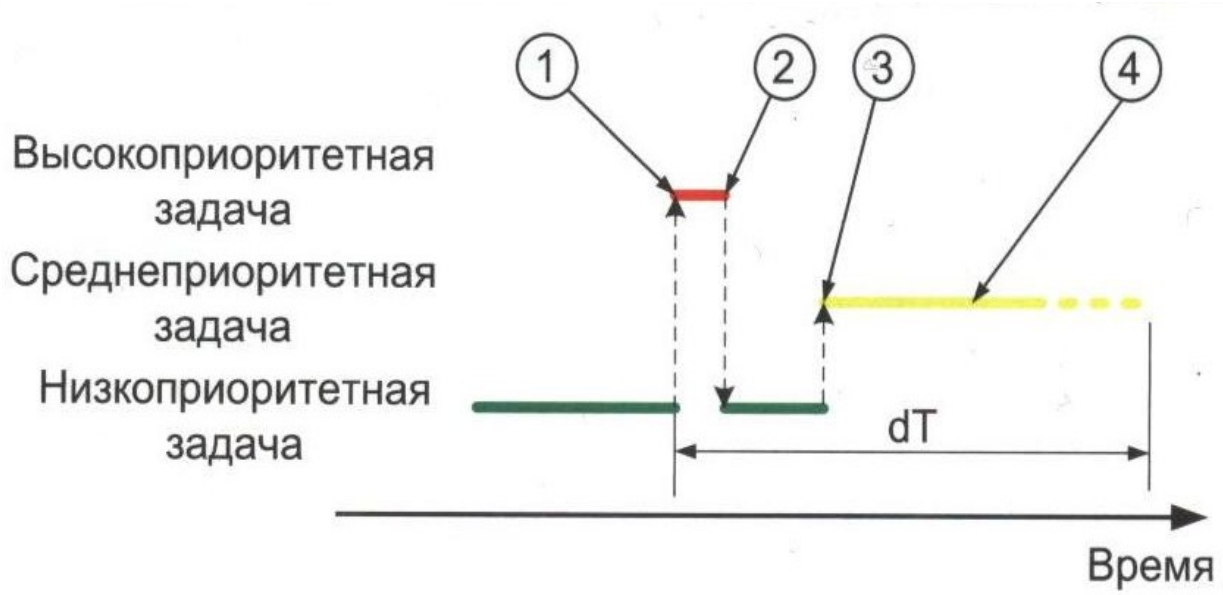
Так как умозрительные решения чреваты ошибками, показатели СРВ при этом снижаются. Чтобы более эффективно осуществить указанное преобразование ограничений, проектировщик может воспользоваться теорией расписаний или имитационным моделированием, хотя и это может оказаться бесполезным. На сегодняшний день не имеется иного решения, поэтому понятие приоритета нити необходимо.

ОСРВ должна поддерживать предсказуемые механизмы синхронизации.

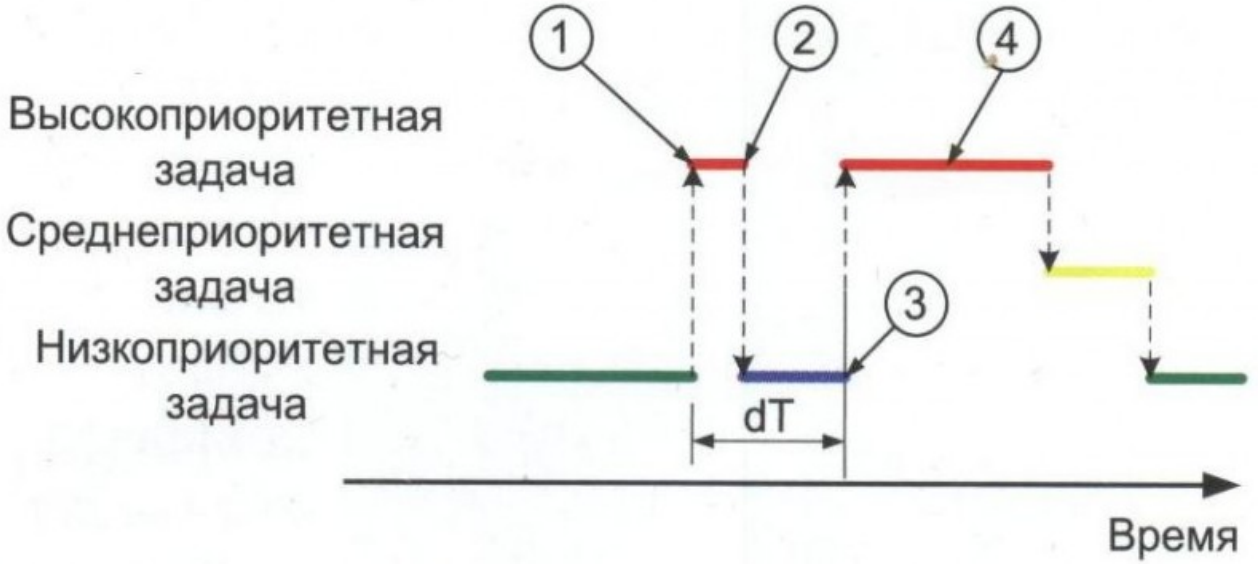
Задачи разделяют данные (ресурсы) и должны сообщаться друг с другом, следовательно, должны существовать механизмы блокирования и коммуникации.

ОСРВ должна обеспечивать механизм наследования приоритетов.

Комбинация приоритета нити и разделение ресурсов между ними приводит к другому явлению: классической проблеме инверсии приоритетов. Чтобы устранить такие инверсии, ОСРВ должна допускать наследование приоритета, т.е. повышение приоритета до уровня вызывающей нити. Наследование означает, что блокирующая ресурс нить наследует приоритет блокируемой нити (справедливо лишь в том случае, если блокируемая нить имеет более высокий приоритет).



Инверсия приоритетов



Наследование приоритетов

Поведение ОСРВ должно быть известным и предсказуемым (задержки обработки прерываний, задержки переключения задач, задержки драйверов и т.д.).

Это значит, что во всех сценариях рабочей нагрузки системы должно быть определено максимальное время отклика.

Характеристики ОСРВ

Время реакции системы на внешние события.

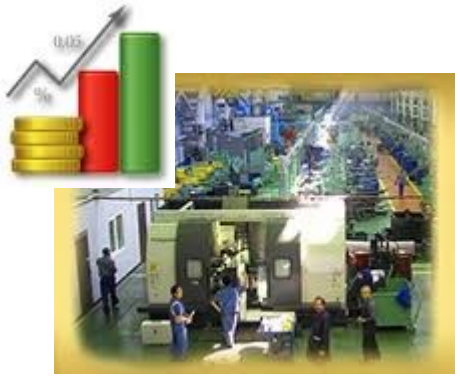
Согласно определению, ОСРВ должна обеспечить требуемый уровень сервиса в заданный промежуток времени. Этот промежуток времени задается обычно периодичностью и скоростью процессов, которым управляет система.

Приблизительное время реакции в зависимости от области применения ОСРВ может быть следующее:

- математическое моделирование- несколько микросекунд
- радиолокация- несколько миллисекунд
- складской учет- несколько секунд
- управление производством - несколько минут

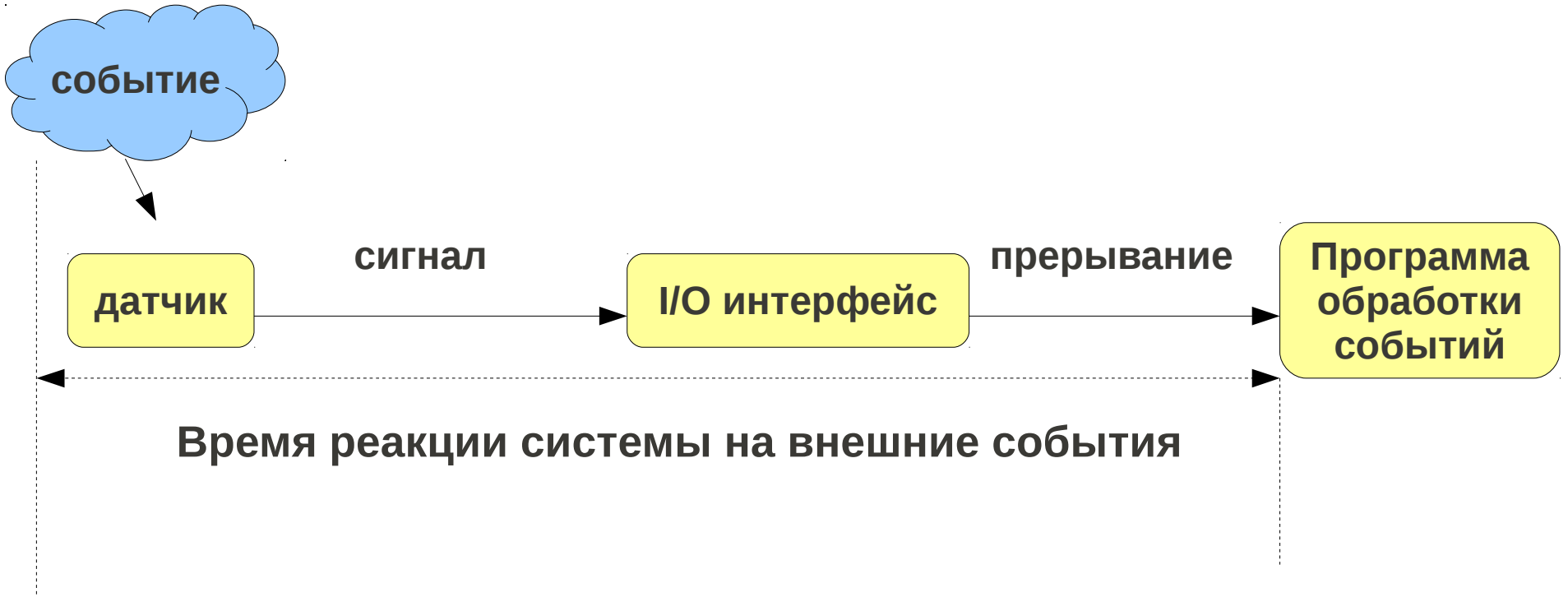
1 микросекунда

1 миллисекунда



1 минута

Время реакции системы на внешние события - интервал времени от события на объекте и до выполнения первой инструкции в программе обработки этого события и является временем реакции системы на события.



Время переключения контекста.

В ОСРВ должна быть заложена возможность одновременной обработки нескольких событий, поэтому все операционные системы реального времени являются многозадачными (многопроцессными, многонитиевыми).

Для того, чтобы уметь оценивать накладные расходы системы при обработке параллельных событий, необходимо знать время, которое система затрачивает на передачу управления от процесса к процессу (от задачи к задаче, от нити к нити), то есть **время переключения контекста.**

Время перезагрузки системы.

Этот параметр важен для систем, от которых требуется непрерывная работа; в этих случаях ставятся ловушки, отслеживающие зависание системы или приложений, и, если таковое произошло, автоматически перезагружающие систему (такие ловушки необходимы в системах повышенной надежности, т.к. от ошибок, по крайней мере, в приложениях никто не застрахован).

В таких случая важным является такое свойство системы как ее живучесть при незапланированных перезагрузках. Большинство операционных систем реального времени устойчивы к перезагрузкам и могут быть прерваны и перезагружены в любое время.

Размеры системы.

Для систем реального времени важным параметром является размер системы исполнения, а именно суммарный размер минимально необходимого для работы приложения системного набора (ядро, системные модули, драйверы и т. д.).

С развитием аппаратного обеспечения значение этого параметра уменьшается, тем не менее он остается важным и производители систем реального времени стремятся к тому, чтобы размеры ядра и обслуживающих модулей системы были невелики.

Примеры: размер ядра операционной системы реального времени OS-9 на микропроцессорах Motorola 68xxx - 22 KB, VxWorks - 16 KB.

16 Кб

32 Кб

Возможность исполнения системы из ПЗУ (ROM).

Система должна иметь возможность осуществлять загрузку из ПЗУ. Для экономии места в ПЗУ часть системы может храниться в сжатом виде и загружаться в ОЗУ по мере необходимости.

Часто система позволяет исполнять код как в ПЗУ, так и в ОЗУ. При наличии свободного места в ОЗУ система может копировать себя из медленного ПЗУ в более быстрое ОЗУ.

Наличие необходимых драйверов устройств

Поддержка процессоров различной архитектуры

Механизмы реального времени

Система приоритетов и алгоритмы диспетчеризации.

Базовыми инструментами разработки сценария работы системы являются система приоритетов процессов (задач) и алгоритмы планирования (диспетчеризации) операционных системах реального времени.

В многозадачных ОС общего назначения используются, как правило, различные модификации алгоритма круговой диспетчеризации, основанные на понятии непрерывного кванта времени ("time slice"), предоставляемого процессу для работы. Планировщик по истечении каждого кванта времени просматривает очередь активных процессов и принимает решение, кому передать управление, основываясь на приоритетах процессов (численных значениях, им присвоенных).

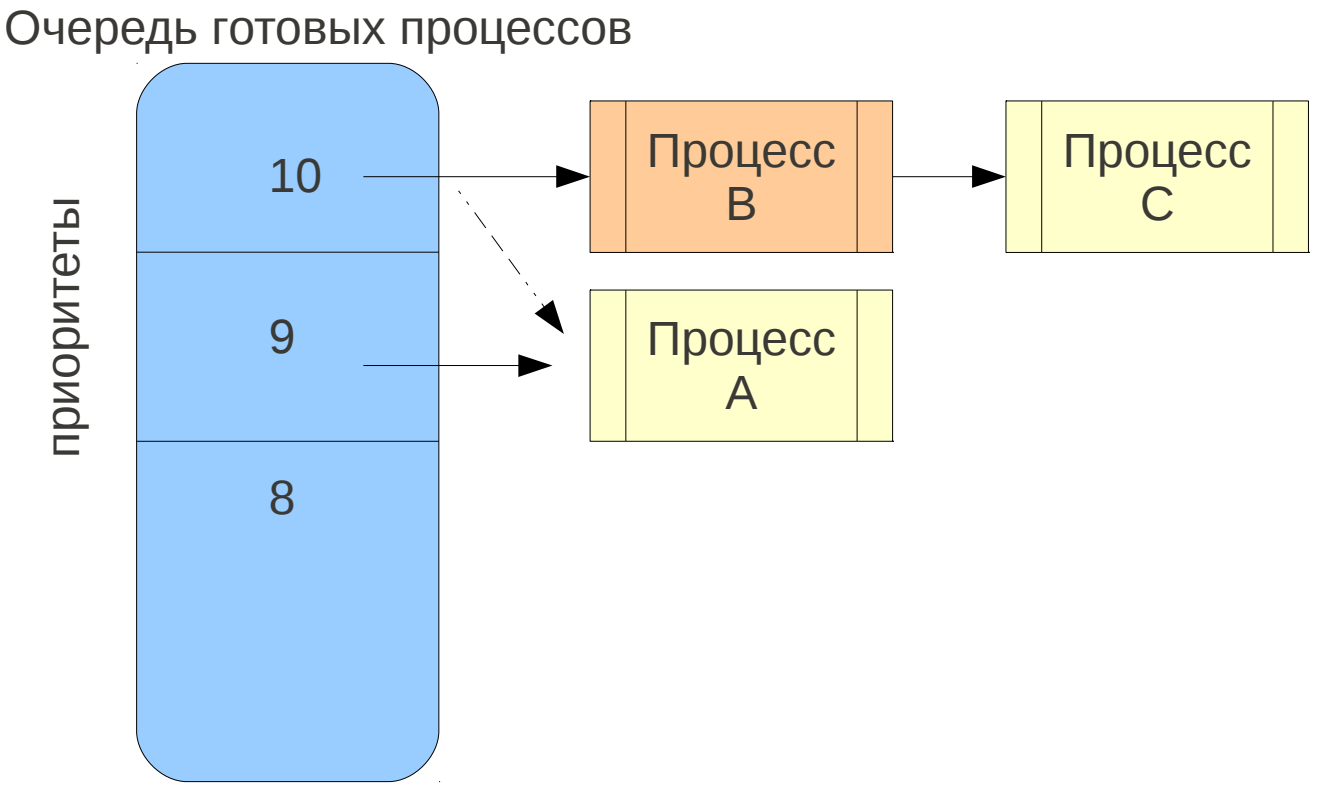
Приоритеты могут быть фиксированными или меняться со временем - это зависит от алгоритмов планирования в данной ОС, но рано или поздно процессорное время получат все процессы в системе.

Алгоритмы круговой диспетчеризации неприменимы в чистом виде в операционных системах реального времени. Основной недостаток - непрерывный квант времени, в течение которого процессором владеет только один процесс.

Планировщики же операционных систем реального времени имеют возможность сменить процесс до истечения "time slice", если в этом возникла необходимость. Один из возможных алгоритмов планирования при этом "приоритетный с вытеснением".

Мир операционных систем реального времени отличается богатством различных алгоритмов планирования: динамические, приоритетные, монотонные, адаптивные и пр., цель же всегда преследуется одна - предоставить инструмент, позволяющий в нужный момент времени исполнять именно тот процесс, который необходим.

Пример алгоритма адаптивной диспетчеризации (QNX)



Если процесс использовал свой квант времени (т.е. он не блокировался), то его приоритет уменьшается на 1. (priority decay), при этом "пониженный" процесс не будет продолжать "снижаться", даже если он использовал еще один квант времени и не блокировался - он снизится только на один уровень ниже своего исходного приоритета.

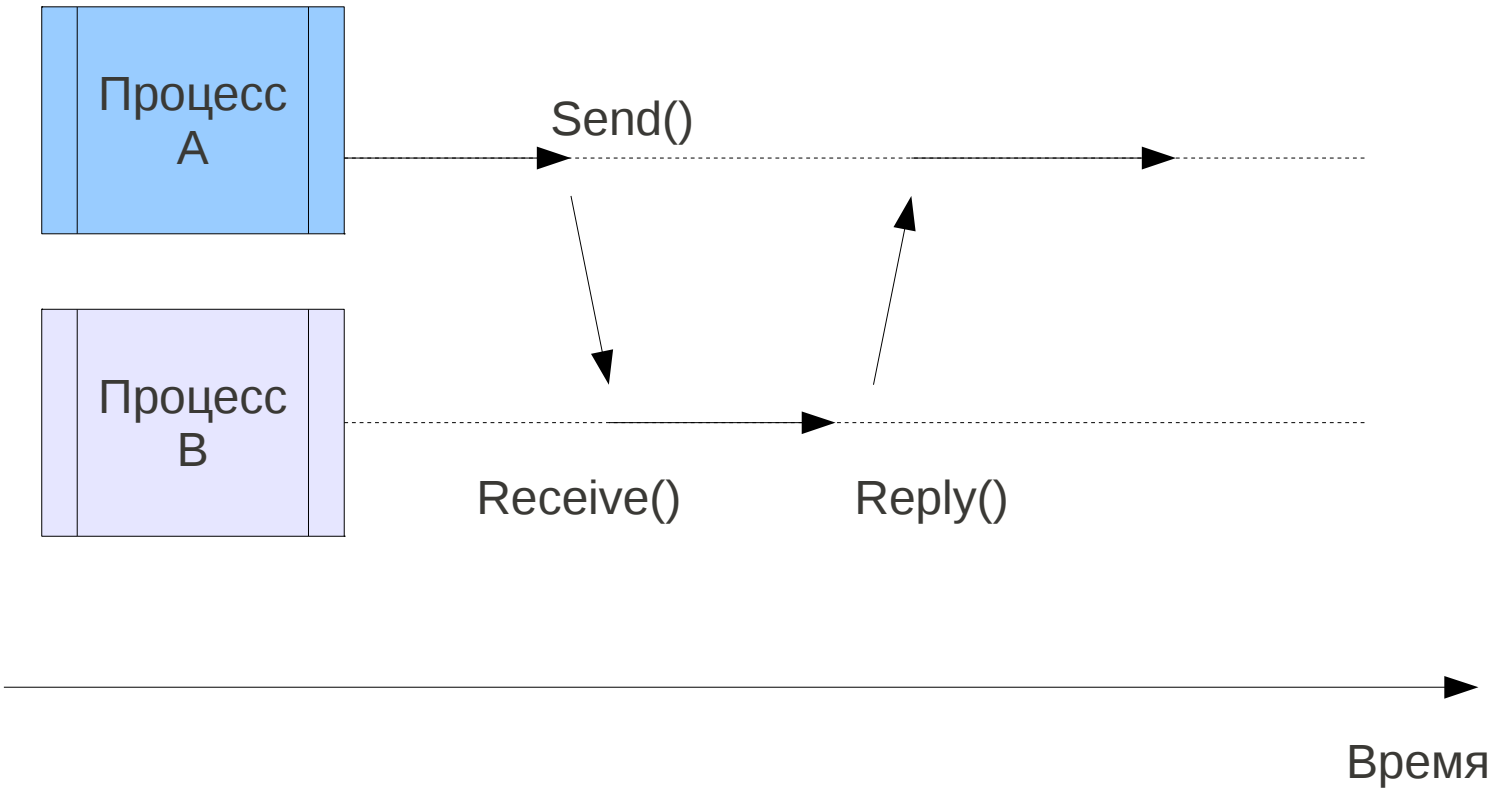
Если процесс блокируется, то ему возвращается первоначальное значение приоритета.

Механизмы межзадачного взаимодействия.

Другой набор механизмов реального времени относится к средствам синхронизации процессов и передачи данных между ними. Для операционных систем реального времени характерна развитость этих механизмов. К таким механизмам относятся: семафоры, мьютексы, события, сигналы, средства для работы с разделяемой памятью, каналы данных (pipes), очереди сообщений.

Многие из подобных механизмов используются и в ОС общего назначения, но их реализация в операционных системах реального времени имеет свои особенности - время исполнения системных вызовов почти не зависит от состояния системы, и в каждой операционной системе реального времени есть по крайней мере один быстрый механизм передачи данных от процесса к процессу.

Пример межзадачного взаимодействия на основе сообщений



Средства для работы с таймерами.

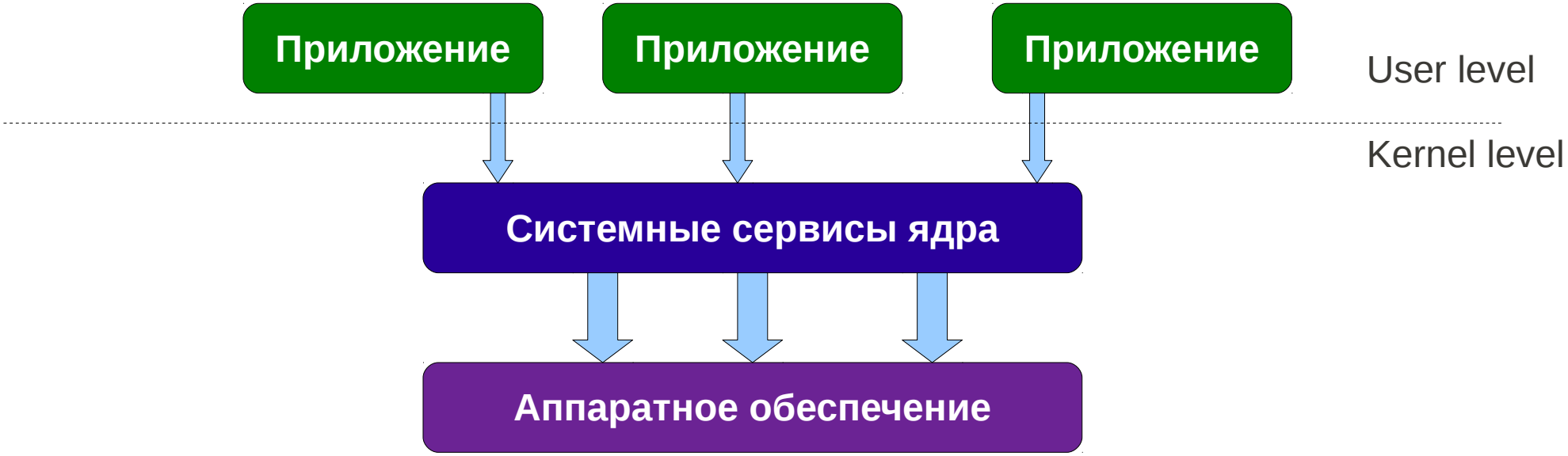
Такие инструменты, как средства работы с таймерами, необходимы для систем с жестким временным регламентом, поэтому развитость средств работы с таймерами - необходимый атрибут операционных систем реального времени. Эти средства, как правило, позволяют:

- измерять и задавать различные промежутки времени (от 1 мкс и выше),
- генерировать прерывания по истечении временных интервалов,
- создавать разовые и циклические будильники

Архитектура операционных системам реального времени

Монолитная архитектура.

ОС определяется как набор модулей, взаимодействующих между собой внутри ядра системы и предоставляющих прикладному ПО входные интерфейсы для обращений к аппаратуре.



Недостатки: плохая предсказуемость поведения, вызванная сложным взаимодействием модулей между собой.

Примеры монолитных ОС:

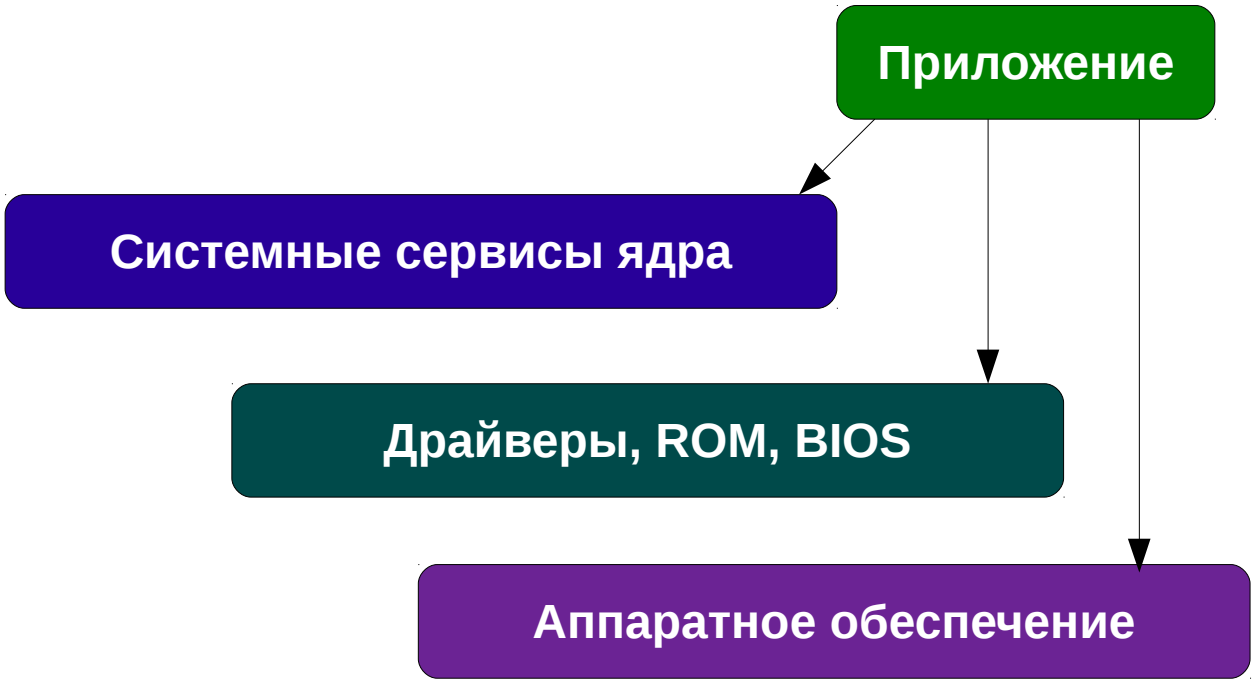
Unix

Novell NetWare

Linux

Уровневая (слоевая) архитектура

Прикладное ПО имеет возможность получить доступ к аппаратуре не только через ядро системы и её сервисы, но и напрямую. По сравнению с монолитной такая архитектура обеспечивает значительно большую степень предсказуемости реакций системы, а также позволяет осуществлять быстрый доступ прикладных приложений к аппаратуре.



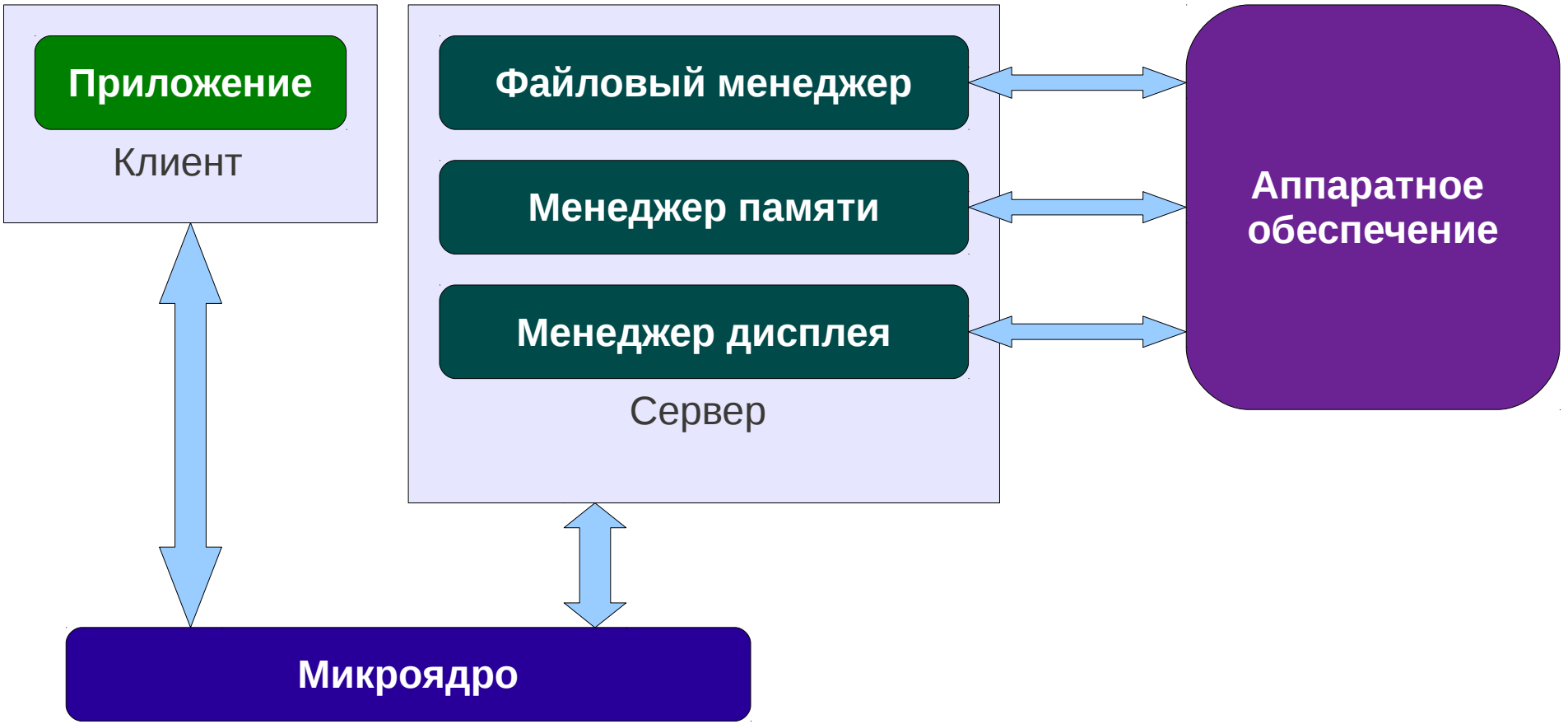
Главным недостатком таких систем является отсутствие многозадачности.

Примеры уровневых ОС:

MS-DOS

Архитектура «клиент-сервер»

Основной принцип архитектуры заключается в вынесении сервисов ОС в виде серверов на уровень пользователя и выполнении микроядром функций диспетчера сообщений между клиентскими пользовательскими программами и серверами — системными сервисами.

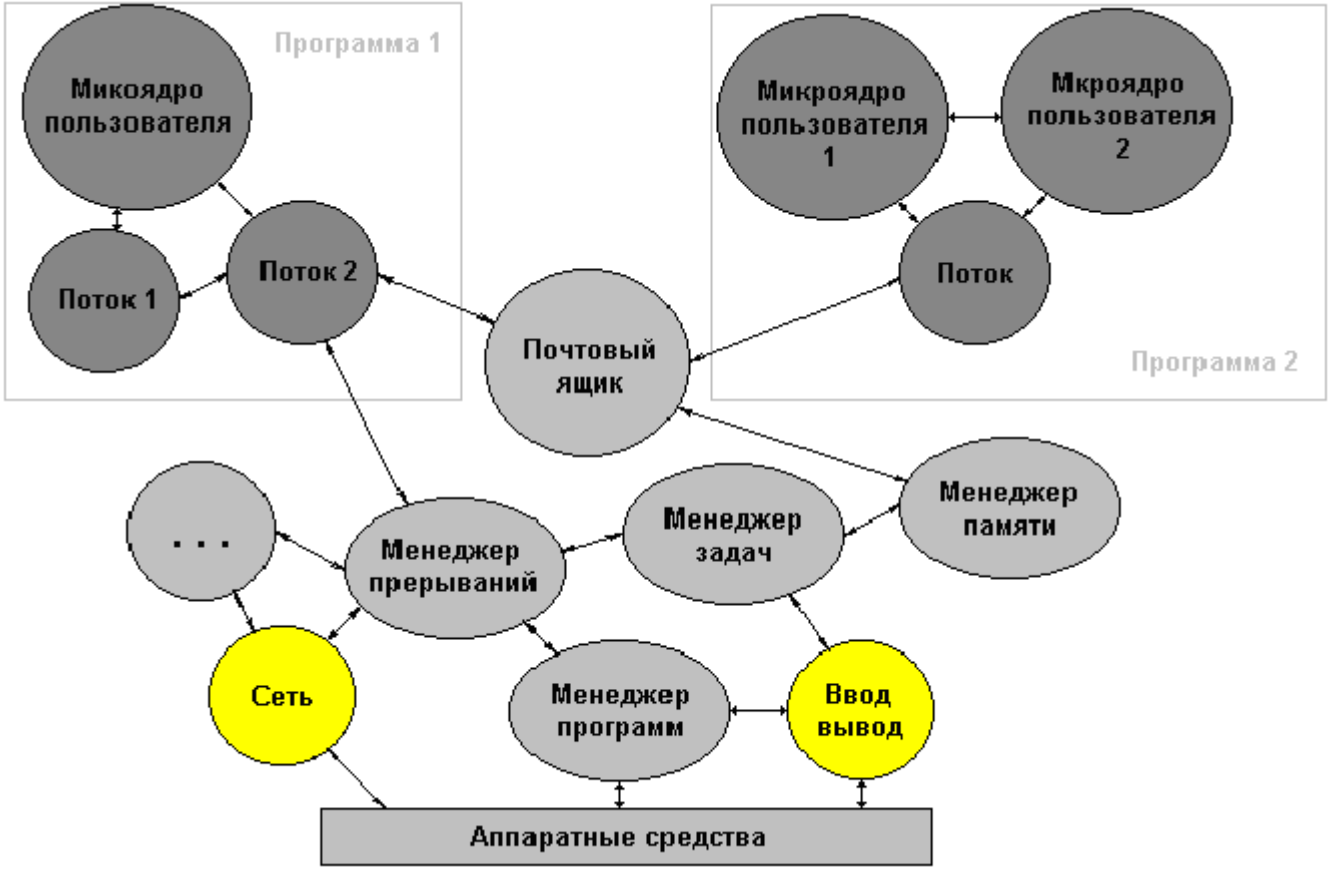


Преимущества архитектуры “клиент-сервер”:

1. Повышенная надёжность, так как каждый сервис является, по сути, самостоятельным приложением и его легче отладить и отследить ошибки;
2. Улучшенная масштабируемость, поскольку ненужные сервисы могут быть исключены из системы без ущерба к её работоспособности;
3. Повышенная отказоустойчивость, так как «зависший» сервис может быть перезапущен без перезагрузки системы.

Объектная архитектура на основе объектов-микроядер.

В этой архитектуре API отсутствует вообще. Взаимодействие между компонентами системы (микроядрами) и пользовательскими процессами осуществляется посредством обычного вызова функций, поскольку и система, и приложения написаны на одном языке (для ОСРВ SoftKernel это C++).Это обеспечивает максимальную скорость системных вызовов.



Фактическое равноправие всех компонент системы обеспечивает возможность переключения задач в любое время, т.е. система полностью preemptible.

Объектно-ориентированный подход обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

Роль API играет компилятор и динамический редактор объектных связей (linker). При старте приложения динамический linker загружает нужные ему микроядра (т.е., в отличие от предыдущих систем, не все компоненты самой операционной системы должны быть загружены в оперативную память).

Если микроядро уже загружено для другого приложения, оно повторно не загружается, а использует код и данные уже имеющегося ядра. Это позволяет уменьшить объем требуемой памяти.

.

Ядро ОСРВ

Функции ядра

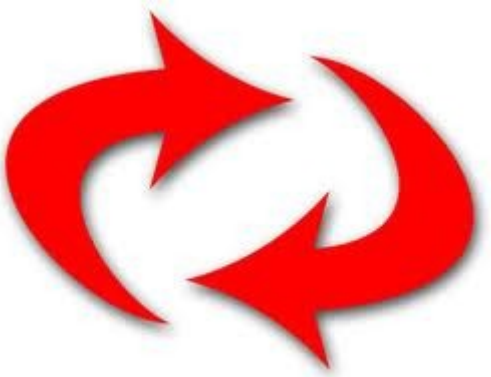
Синхронизация ресурсов. Метод синхронизации требует ограничить доступ к общим ресурсам (данным и внешним устройствам). Наиболее распространенный тип примитивной синхронизации - двоичный семафор, обеспечивающий избирательный доступ к общим ресурсам. Так, процесс, требующий защищенного семафором ресурса, вынужден ожидать до тех пор, пока семафор не станет доступным, что свидетельствует об освобождении ожидаемого ресурса, и, захватив ресурс, установить семафор. В свою очередь, другие процессы также будут ожидать доступа к ресурсу вплоть до того момента, когда семафор возвратит соответствующий ресурс системе распределения ресурсов.

Системы, обладающие большей ошибкоустойчивостью могут иметь счетный семафор. Этот вид семафора разрешает одновременный доступ к ресурсу лишь определенному количеству процессов.

Межзадачный обмен. Часто необходимо обеспечить передачу данных между программами внутри одной и той же системы Кроме того, во многих приложениях возникает необходимость взаимодействия с другими системами через сеть.

Внутренняя связь может быть осуществлена через систему передачи сообщений.

Внешнюю связь можно организовать либо через датаграмму (наилучший способ доставки), либо по линиям связи (гарантированная доставка). Выбор того или иного способа зависит от протокола связи.



Разделение данных. В прикладных программах, работающих в реальном времени, наиболее длительным является сбор данных. Данные часто необходимы для работы других программ или нужны системе для выполнения каких-либо своих функций.

Во многих системах предусмотрен доступ к общим разделам памяти. Широко распространена организация очереди данных. Применяется много типов очередей, каждый из которых обладает собственными достоинствами.

Обработка запросов внешних устройств. Каждая прикладная программа в реальном времени связана с внешним устройством определенного типа. Ядро должно обеспечивать службы ввода/вывода, позволяющие прикладным программам осуществлять чтение с этих устройств и запись на них.

Для приложений реального времени обычным является наличие специфического для данного приложения внешнего устройства. Ядро должно предоставлять сервис, облегчающий работу с драйверами устройств. Например, давать возможность записи на языках высокого уровня - таких, как Си или Паскаль.

Обработка особых ситуаций. Особая ситуация представляет собой событие, возникающее во время выполнения программы. Она может быть синхронной, если ее возникновение предсказуемо, как, например, деление на нуль. А может быть и асинхронной, если возникает непредсказуемо, как, например, падение напряжения.

Предоставление возможности обрабатывать события такого типа позволяет прикладным программам реального времени быстро и предсказуемо отвечать на внутренние и внешние события.

Существуют два метода обработки особых ситуаций - использование значений состояния для обнаружения ошибочных условий и использование обработчика особых ситуаций для прерывания ошибочных условий и их корректировки.

Стандартизованные профили систем реального времени - POSIX.13

Для систем реального времени в стандартах POSIX определяется четыре профиля, соответствующие четырем типам систем реального времени различной функциональности, начиная от простых встроенных систем с минимальными возможностями и, кончая многоцелевыми вычислительными системами с расширенными возможностями, которые применяются, например, в контурах управления крупными военными комплексами, сложными технологическими процессами и т.п.

Стандартизованными POSIX-профилями реального времени являются:

профиль минимальных (встроенных) систем реального времени (Minimal (Embedded) Realtime System Profile) - PSE51;

профиль систем-контроллеров реального времени (Realtime Controller System Profile) - PSE52;

профиль выделенных систем реального времени (Dedicated Realtime System Profile) - PSE53;

профиль многоцелевых систем реального времени (Multi-Purpose Realtime System Profile) - PSE54.

Стандарты ОСРВ

POSIX

(IEEE Portable Operating System Interface for Computer Environments, IEEE 1003.1)

Стандарт POSIX был создан как стандартный интерфейс сервисов операционных систем.

Этот стандарт дает возможность создавать переносимые приложения. Впоследствии этот стандарт был расширен особенностями режима реального времени

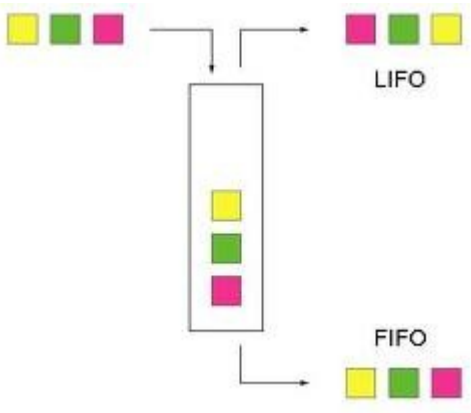
Несмотря на то, что стандарт POSIX вырос из Unix, он затрагивает основополагающие абстракции операционных систем, а расширения реального времени применимы ко всем ОСРВ.

К настоящему времени стандарт POSIX рассматривается как семейство родственных стандартов: IEEE Std 1003.n (где n – это номер).

Стандарт 1003.1b (Realtime Extensions) содержит расширения реального времени:

А. Диспетчеризация процессов реального времени.

SCHED_FIFO SCHED_RR SCHED_OTHER



B. Блокирование виртуальной памяти.

Хотя в базовом стандарте POSIX использование виртуальной памяти не требуется, в UNIX-системах этот механизм широко распространен. Он очень эффективен при работе программ, не относящихся к программам реального времени, но приводит к непредсказуемости времени реакции системы.

Для того чтобы ограничить время доступа к памяти, в стандарте 1003.1b определяются функции блокировки (фиксации) в памяти всего адресного пространства процесса или отдельных его областей.

Эти функции следует использовать для критичных ко времени процессов, а также для процессов, с которыми синхронизируются критичные ко времени процессы. В этом случае время их реакции может быть предсказуемым.

Xenomai

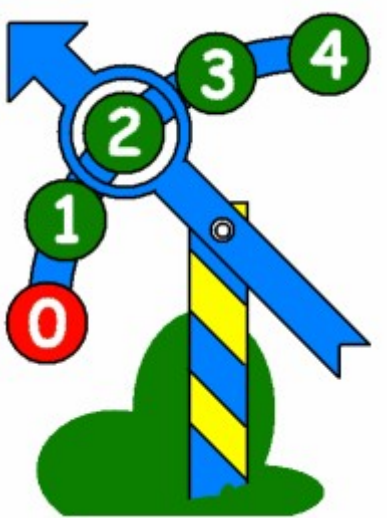
```
/* Avoids memory swapping for this program */

  mlockall(MCL_CURRENT|MCL_FUTURE);
```

C. Синхронизация процессов.

В стандарте 1003.1b определяются функции управления синхронизацией процессов с помощью семафоров-счетчиков.

Эти семафоры идентифицируются по имени, находящемся в некотором пространстве имен, определяемом при реализации стандарта. Это пространство имен может совпадать, но не обязательно, с пространством имен файлов.

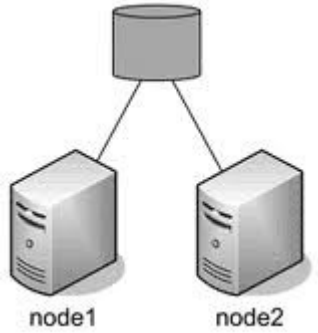
Семафор-счетчик – это общий механизм синхронизации, который позволяет реализовать взаимно исключающий доступ к разделяемым ресурсам, передачу сигналов, ожидание процессов и другие механизмы синхронизации.



D. Разделяемая память.

В соответствии с базовым стандартом POSIX процессы имеют независимые адресные пространства, но во многих приложениях реального времени (и других) требуется совместное использование, с очень малыми издержками, большого количества данных.

Это возможно в случае, если процессы могут разделять части физической памяти. В стандарте 1003.1b определяются объекты разделяемой памяти



E. Сигналы реального времени.

Сигналы реального времени устанавливаются в очередь, поэтому события не теряются.

Необработанные сигналы реального времени извлекаются из очереди по приоритетам, где в качестве приоритета служит номер сигнала. Это предоставляет возможность быстрого отклика на события, требующие неотложной реакции.

Сигналы реального времени содержат дополнительное поле данных, которое может использоваться прикладной системой для обмена между генератором сигнала и его обработчиком. Например, это поле данных может использоваться для идентификации источника сигнала.

F. Взаимодействие процессов.

Очереди сообщений идентифицируются по имени, принадлежащему некоторому пространству имен, определяемому при реализации стандарта.

Сообщения имеют связанное с ними поле приоритета и извлекаются из очереди в соответствии с приоритетом.

Передача и получение сообщений может блокироваться и разблокироваться.

Передача и получение не синхронизируются, то есть, отправитель не ждет, когда получатель действительно извлечет сообщение из очереди.

Максимальный размер сообщений и очередей определяется пользователем, а необходимые для поддержания очереди ресурсы могут выделяться заранее во время разработки приложения, что повышает предсказуемость работы с очередями сообщений.

G. Часы и таймеры.

Часы реального времени должны обеспечивать разрешение минимум 20 мс.

Для отсчета временных интервалов на основе часов реального времени или других часов, определенных при реализации стандарта, могут создаваться таймеры.

По истечении заданного интервала времени эти таймеры генерируют сигнал, направленный процессу, создавшему данный таймер.

Существует несколько опций, таких, как периодическая сигнализация, единичный сигнал и т. д., которые позволяют легко реализовать, например, генерацию периодических событий.

H. Асинхронный ввод/вывод

В стандарте 1003.1b определяются функции, которые обеспечивают возможность совмещать прикладную обработку и операции ввода/вывода, инициированные данным приложением.

Асинхронные операции ввода/вывода подобны обычным операциям, за исключением того, что после того как процесс инициировал асинхронную операцию ввода/вывода, он продолжает выполняться параллельно этой операции.

Когда операция завершается, данному приложению может быть послан сигнал.

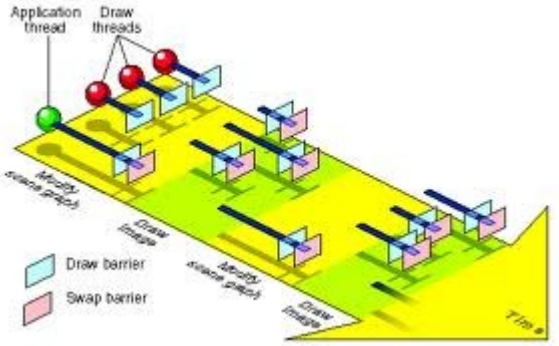
Стандарт 1003.1c (Threads) касается функций поддержки многопоточной обработки внутри процесса:

A. Управление потоками.

Определяются функции создания потока, ожидания завершения потока, нормального завершения потока, открепления потока (то есть указания разработчику, что связанная с потоком память может после завершения потока перераспределяться) или создания конкретной потока только в том случае, если он еще не был создана.

Другие функции позволяют управлять идентификаторами потоков.

Определяются также функции для управления атрибутами при создании потоков, такими, как размер стека, возможность перераспределения памяти, занимаемой потоком после ее создания, и т.п.



B. Диспетчеризация потоков.

Для потоков определяются те же методы диспетчеризации, что и для процессов в стандарте 1003.1b.

Так как в системе могут одновременно существовать два планировщика (диспетчера) - планировщик процессов и планировщик потоков - определяется концепция конкурентного пространства.

Конкурентное пространство некоторого потока - это набор потоков, с которыми он конкурирует за центральный процессор.

C. Синхронизация потоков

Для потоков определяется два примитива синхронизации: мьютексы (mutex, mutually exlusive) и условные переменные (condvars, condition variables)

.

Мьютексы используются для синхронизации взаимно исключающего доступа потоков к разделяемым ресурсам, а условные переменные используются для сигнализации и ожидания событий среди потоков.

Ожидание условной переменной, с которой должен быть связан сигнал, можно задать с помощью таймаута.

DO-178B

Стандарт DO-178B, создан Радиотехнической комиссией по аэронавтике (RTCA, Radio Technical Commission for Aeronautics) для разработки ПО бортовых авиационных систем.

Стандартом предусмотрено пять уровней серьезности отказа, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать работоспособность всей системы в целом при возникновении отказов данного уровня серьезности

Данный стандарт определяет следующие уровни сертификации:

- A (катастрофический),
- B (опасный),
- C (существенный),
- D (несущественный)
- E (не влияющий).

До тех пор пока все жесткие требования этого стандарта не будут выполнены, вычислительные системы, влияющие на безопасность, никогда не поднимутся в воздух.

ARINC-653

Стандарт ARINC-653 (Avionics Application Software Standard Interface) разработан компанией ARINC в 1997 г.

Этот стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между ОС авиационного компьютера и прикладным ПО.

Требования к интерфейсу между прикладным ПО и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному ПО контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов.

В 2003 г. принята новая редакция этого стандарта. ARINC-653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин.

Планирование задач

Необходимость планирования задач появляется, как только в очереди активных (готовых) задач появляются более одной задачи (в многопроцессорных системах - более числа имеющихся процессоров).

Алгоритм планирования задач является основным отличием систем реального времени от "обычных" операционных систем.

В ОС общего назначения целью планирования является обеспечение выполнения всех задач из очереди готовых задач, обеспечивая иллюзию их параллельной работы и не допуская монополизацию процессора какой-либо из задач.

В ОСРВ же целью планирования является **обеспечение выполнения каждой готовой задачи к определенному моменту времени**, при этом часто "параллельность" работы задач не допускается, поскольку тогда время исполнения задачи будет зависеть от наличия других задач.

Планировщик задач (scheduler) - это модуль (программа), отвечающий за разделение времени имеющихся процессоров между выполняющимися задачами. Отвечает за коммутацию задач из состояния блокировки в состояние готовности, и за выбор задачи (задач - по числу процессоров) из числа готовых для исполнения процессором (ами).

Ключевым вопросом планирования является выбор момента принятия решения

Алгоритмы планирования можно разделить на две категории согласно их поведению после прерываний.

Алгоритмы планирования **без переключений**, иногда называемые также **неприоритетным** планированием, выбирают процесс и позволяют ему работать вплоть до блокировки либо вплоть до того момента, когда процесс сам не отдаст процессор.

Процесс не будет прерван, даже если он работает часами. Соответственно, решения планирования не принимаются по прерываниям от таймера.

После обработки прерывания таймера управление всегда возвращается приостановленному процессу.

Алгоритмы планирования **с переключениями**, называемые также **приоритетным** планированием, выбирают процесс и позволяют ему работать некоторое максимально возможное время.

Если к концу заданного интервала времени процесс все еще работает, он приостанавливается и управление переходит к другому процессу.

Приоритетное планирование требует прерываний по таймеру, происходящих в конце отведенного периода времени (решения планирования могут, например, приниматься при каждом прерывании по таймеру, или при каждом k-ом прерывании), чтобы передать управление планировщику.

Приоритетом называется число, приписанное операционной системой (а именно, планировщиком задач) каждому процессу и задаче.

Существуют несколько схем назначения приоритетов.

Фиксированные приоритеты - приоритет задаче назначается при ее создании и не меняется в течение ее жизни.

Эта схема с различными дополнениями применяется в большинстве систем реального времени. В схемах планирования ОСРВ часто требуется, чтобы приоритет каждой задачи был уникальным.

Турнирное определение приоритета - приоритет последней исполнявшейся задачи понижается.

Определение приоритета по алгоритму **round robin** - приоритет задачи определяется ее начальным приоритетом и временем ее обслуживания.

Чем больше задача обслуживается процессором, тем меньше ее приоритет (но не опускается ниже некоторого порогового значения).

Эта схема в том или ином виде применяется в большинстве UNIX систем.

Алгоритм диспетчеризации FIFO.

Является алгоритмом планирования без переключений. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают.

При FIFO диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

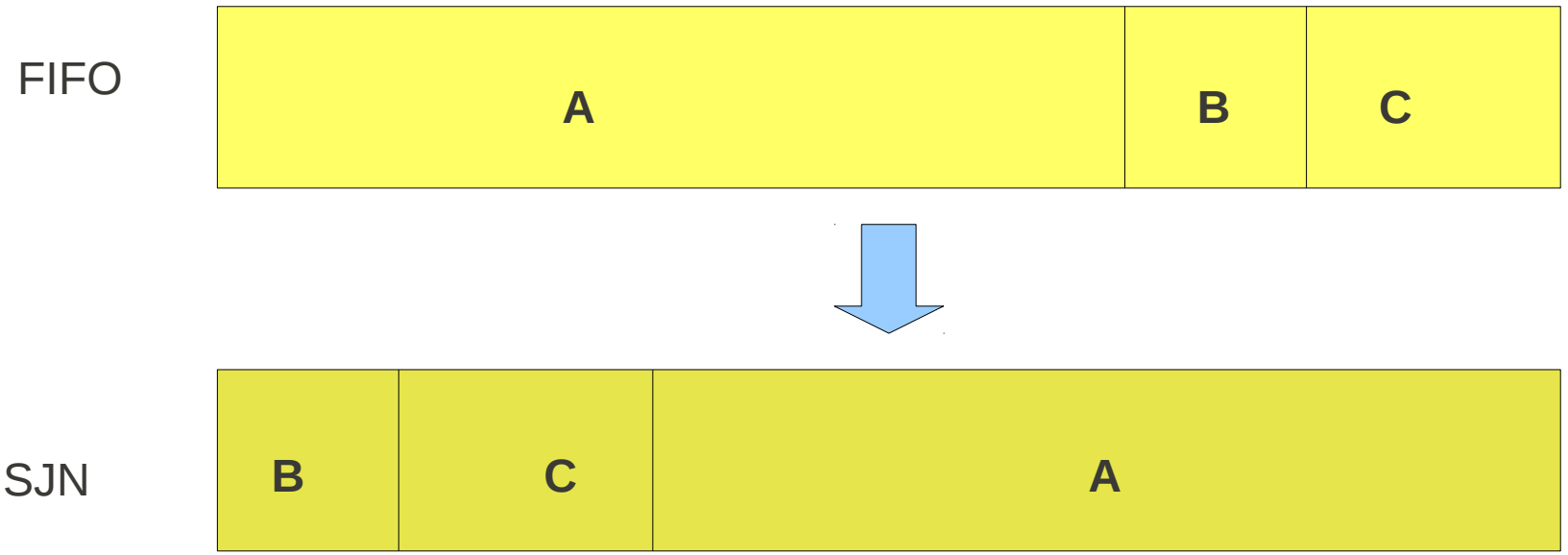
- добровольно уступает управление (заканчивается, блокируется и т.п.);
- вытесняется процессом с более высоким приоритетом.

При отсутствии второго условия возможен случай, когда высокоприоритетная задача будет ожидать окончания работы низкоприоритетной.

«Кратчайшая задача – первая»

Shortest job next (SJN) Shortest Job First (SJF) Shortest Process Next (SPN)

Этот алгоритм без переключений предполагает, что временные отрезки работы известны заранее. В этом алгоритме первым выбирается не самая первая, а самая короткая задача.



Наименьшее оставшееся время выполнения».

В соответствии с этим алгоритмом планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения.

В этом случае также необходимо знать заранее время выполнения каждого процесса.

Когда поступает новый процесс, его полное время выполнения сравнивается с оставшимся временем выполнения текущего процесса. Если время выполнения нового процесса меньше, текущий процесс приостанавливается и управление передается новому процессу.

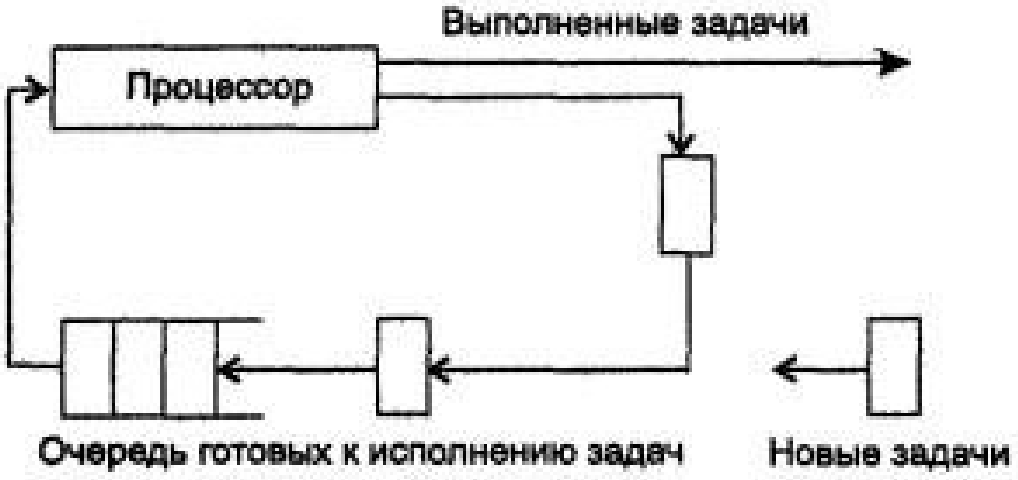
Эта схема позволяет быстро обслуживать короткие процессы.

«Карусельная диспетчеризация (циклическое планирование)».

При карусельной диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

- добровольно уступает управление (т.е. блокируется);
- вытесняется процессом с более высоким приоритетом;
- использовал свой квант времени (timeslice).

После того, как процесс использовал свой квант времени, управление передается следующему процессу, который находится в состоянии готовности и имеет такой же уровень приоритета.



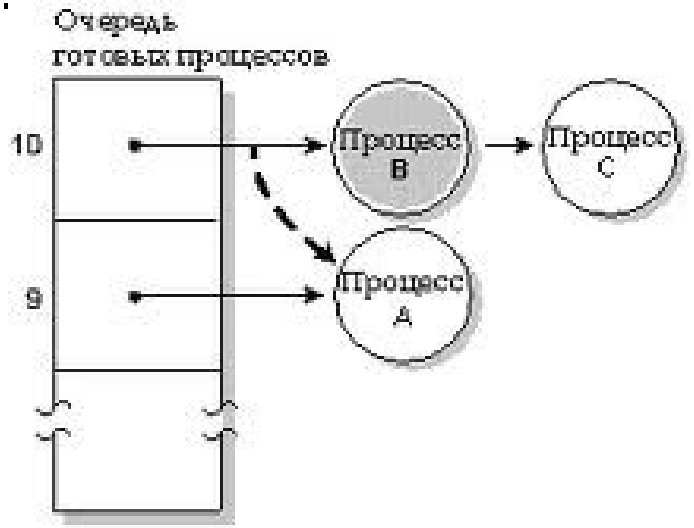
«Адаптивная диспетчеризация».

При адаптивной диспетчеризации процесс ведет себя следующим образом:

Если процесс использовал свой квант времени (т.е. он не блокировался), то его приоритет уменьшается на 1. Это получило название снижение приоритета (priority decay).

"Пониженный" процесс не будет продолжать "снижаться", даже если он использовал еще один квант времени и не блокировался - он снизится только на один уровень ниже своего исходного приоритета.

Если процесс блокируется, то ему возвращается первоначальное значение приоритета.



Планирование периодических процессов

Внешние события, на которые система реального времени должна реагировать, можно разделить на *периодические* (возникающие через регулярные промежутки времени) и *непериодические* (возникающие непредсказуемо).

Возможно наличие нескольких потоков событий, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события.

Если в систему поступает *m* периодических событий, событие с номером *i* поступает с периодом *Pi* и на его обработку уходит *Ci* секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{m} \frac{Ci}{Pi} =< 1$$

Система реального времени, удовлетворяющая этому условию, называется **поддающейся планированию** или **планируемой**.

Соотношение $\frac{Ci}{Pi}$ является частью процессорного времени, используемого процессом I

$\sum_{i=1}^{m} \frac{Ci}{Pi}$ - **коэффициент использования** (или коэффициент загруженности) процессора, который, не может быть больше 1.

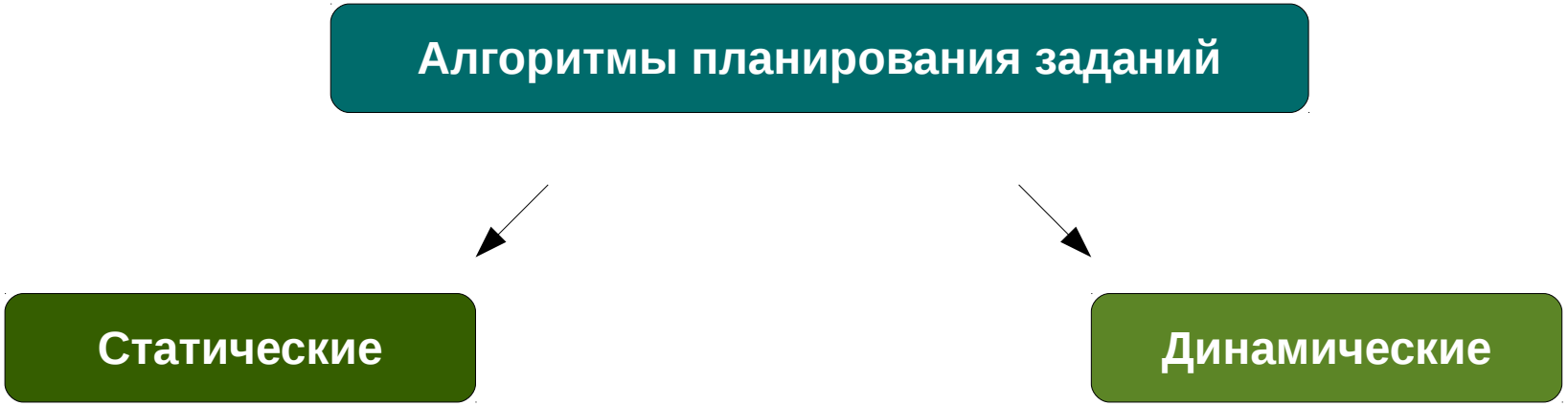
Пример:

Рассмотрим систему с тремя периодическими сигналами с периодами 100, 200, 500 мс соответственно.

Если на обработку этих сигналов уходит 50, 30, 100 мс, система является поддающейся планированию, поскольку 0,5+0,15+0,2<1.

Даже при добавлении четвертого сигнала с периодом в 1 с системой все равно можно будет управлять при помощи планирования, пока время обработки сигнала не будет превышать 150 мс.

Эти расчеты не являются абсолютно верными, так как не учитывают время переключения контекста и не учитывает возникновение непериодических событий.



Статические алгоритмы определяют приемлемый план выполнения заданий по их априорным характеристикам, динамический алгоритм модифицирует план во время исполнения заданий.

Издержки на статическое планирование низки, но оно крайне нечувствительно и требует полной предсказуемости той системы реального времени, на которой оно установлено.

Динамическое планирование связано с большими издержками, но способно адаптироваться к меняющемуся окружению.

Алгоритмы планирования будем рассматривать на примере 3 периодических процессов A, B, C.

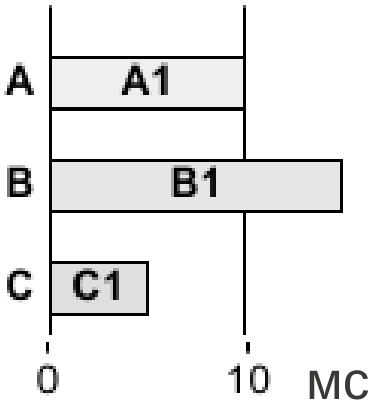
Предположим, что процесс A запускается с периодом 30 мс и временем обработки 10 мс.

Процесс B имеет период 40 мс и время обработки 15 мс.

Процесс C запускается каждые 50 мс и обрабатывается за 5 мс.

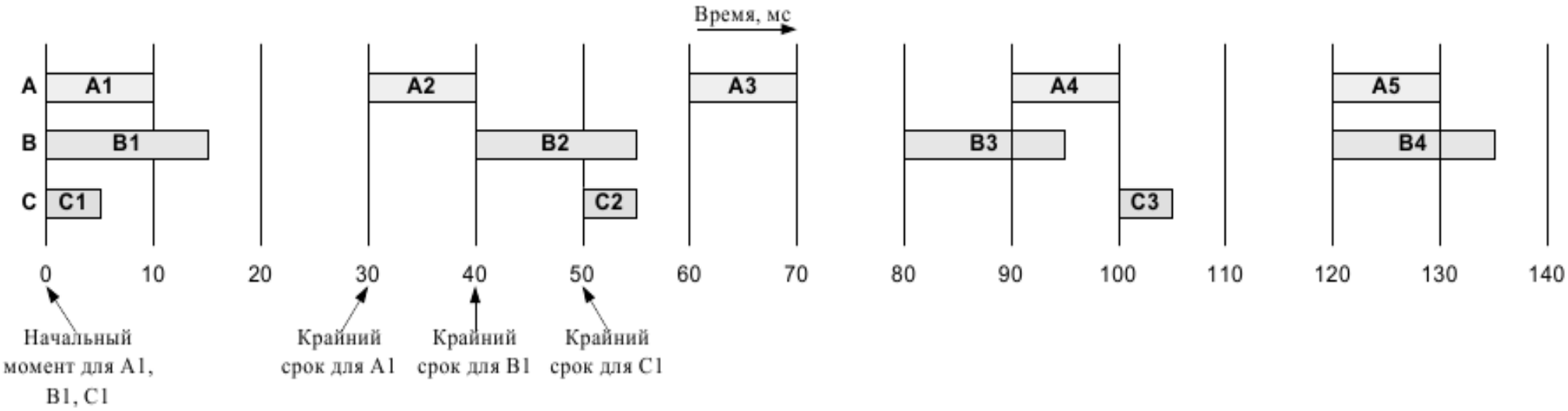
Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы.

Соответственно, система в данном примере поддается планированию.



На рисунке представлена *временная диаграмма* работы процессов.

Видно, что необходимо применить некоторый алгоритм планирования, так как в определенные моменты времени имеется сразу несколько готовых к выполнению процессов.



Алгоритм RMS (Rate Monotonic Scheduling)

Статический алгоритм планирования реального времени для прерываемых периодических процессов - алгоритм RMS (Rate Monotonic Scheduling – планирование с приоритетом, пропорциональным частоте).

Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода

2. Ни один процесс не должен зависеть от любого другого процесса

3. Каждому процессу требуется одинаковое процессорное время на каждом интервале

4. У непериодических процессов нет жестких сроков

5. Прерывание процесса происходит мгновенно, без накладных расходов.

Алгоритм RMS работает, назначая каждому процессу *фиксированный приоритет, обратно пропорциональный периоду* и, соответственно, прямо пропорциональный частоте возникновения событий процесса.

Например, процесс A запускается каждые 30 мс (33 раза в секунду) и получает приоритет 33.

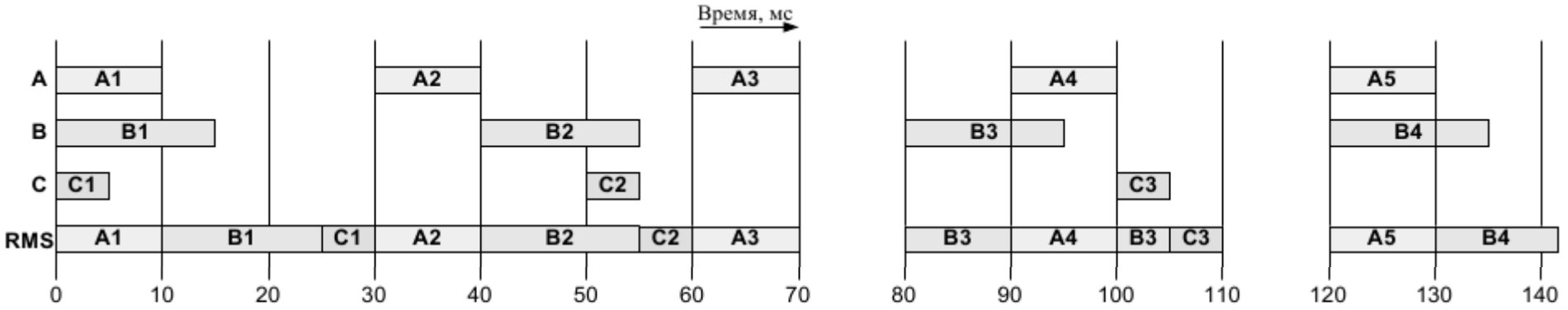
Процесс B запускается каждые 40 мс (25 раз в секунду) и получает приоритет 25.

Процесс C запускается каждые 50 мс (20 раз в секунду) и получает приоритет 20.

Реализация алгоритма требует, чтобы у всех процессов были разные приоритеты.

Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим приоритетом, прерывая при необходимости работающий процесс с меньшим приоритетом.

Таким образом, в нашем примере процесс A может прервать процессы B и C, процесс B может прервать C. Процесс C всегда вынужден ждать, пока процессор не освободится.



0 мс – все процессы готовы, запускается процесс A (с более высоким приоритетом)

70 мс – простой системы (нет готовых процессов)

90 мс – переключение с процесса B на процесс A

Алгоритм RMS **гарантированно** работает в любой системе периодических процессов при условии

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \; " \; < \; m(2^{1/m} - 1)$$

Алгоритм EDF

(Earliest Deadline First – процесс с ближайшим сроком завершения в первую очередь).

Алгоритм EDF представляет собой *динамический* алгоритм, не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования процессора.

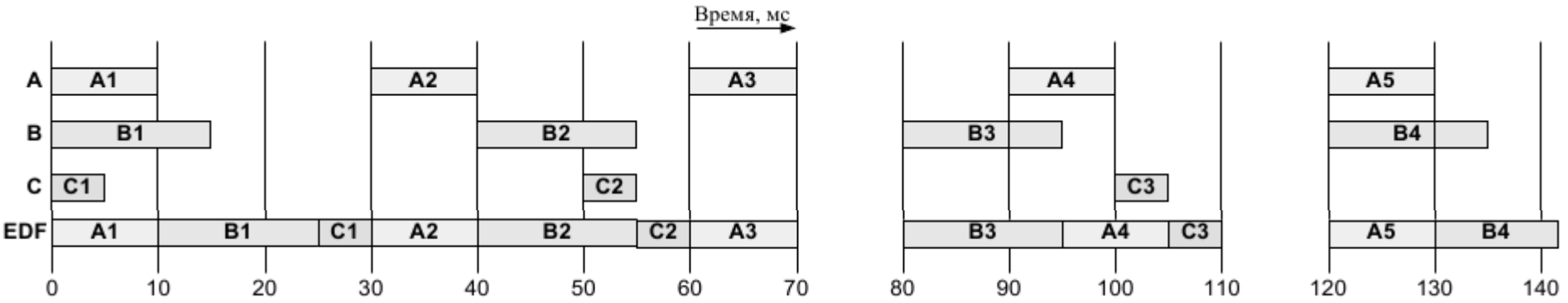
Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания.

Планировщик хранит список процессов, сортированный по срокам выполнения заданий.

Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения.

Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса.

Если у нового процесса график более жесткий, он прерывает работу текущего процесса.



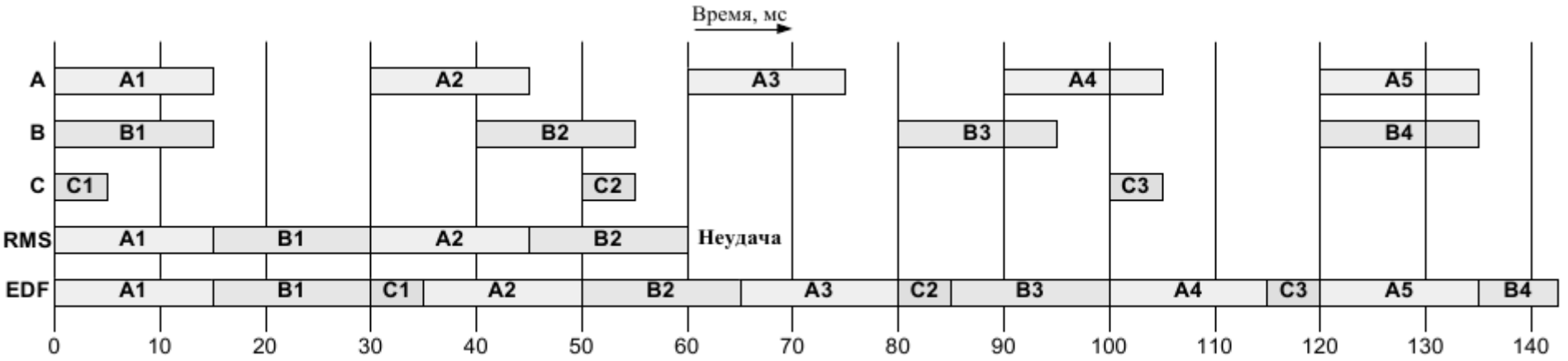
Вначале все процессы находятся в состоянии готовности. Они запускаются в порядке своих крайних сроков (deadline).

Процесс A должен быть выполнен к моменту времени 30, процесс B должен закончить работу к моменту времени 40, процесс C – 50. Таким образом, процесс A запускается первым.

Вплоть до момента времени 90 выбор алгоритма EDF не отличается от RMS.

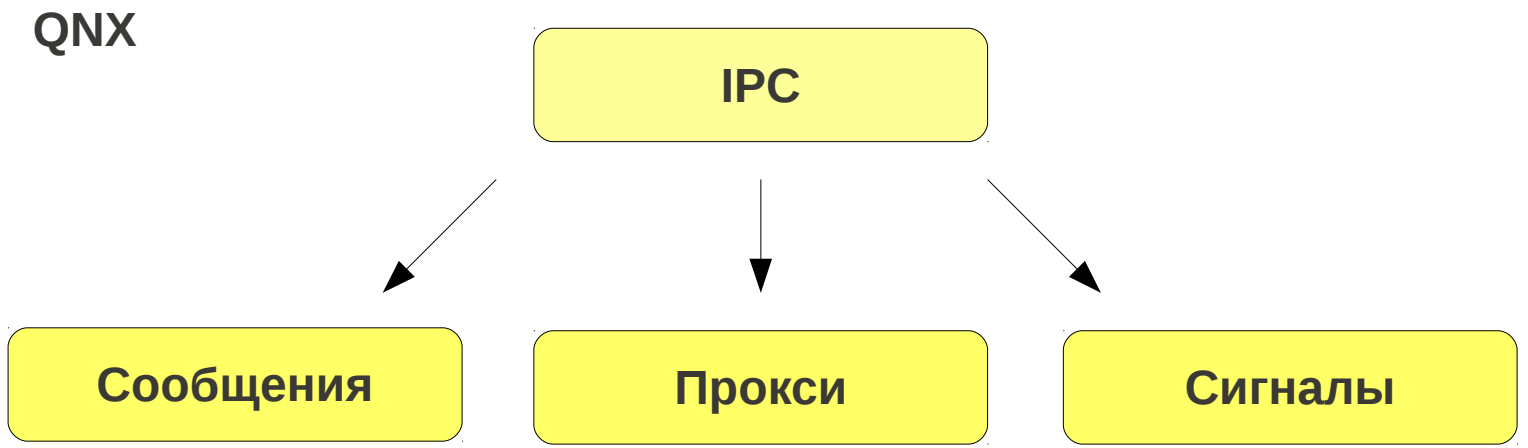
В момент времени 90 процесс A переходит в состояние готовности с тем же крайним сроком выполнения 120, что и у процесса B. Планировщик имеет право выбрать любой из процессов, но поскольку с прерыванием процесса B не связано никаких накладных расходов, лучше предоставить возможность продолжать работу этому процессу.

Сравнение RMS и EDF



Межпроцессное взаимодействие - IPC

В соответствие со стандартом POSIX для операционных систем реального времени определено множество различных форм взаимодействия между процессами и синхронизации процессов: разделяемая память, *семафоры, мьютексы, сигналы, сообщения*.



Сообщения - это основополагающая форма IPC в QNX. Они обеспечивают синхронную связь между взаимодействующими процессами, когда процессу, посылающему сообщение, требуется получить подтверждение того, что оно получено и, возможно, ответ.

Прокси - это особый вид сообщения. Они больше всего подходят для извещения о наступлении какого-либо события, когда процессу, посылающему сообщение, не требуется вступать в диалог с получателем.

Сигналы - это традиционная форма IPC. Они используются для асинхронной связи между процессами.

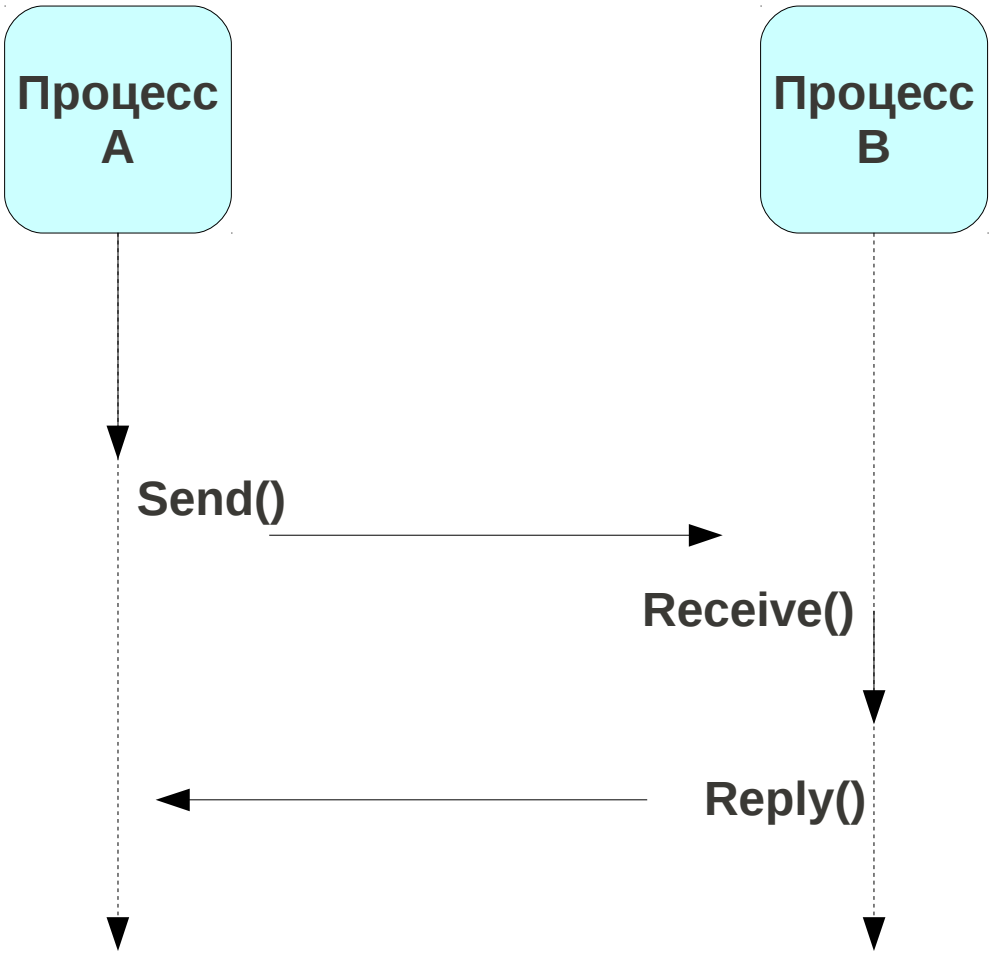
Сообщения в QNX - это пакеты байт, которые синхронно передаются от одного процесса к другому. QNX при этом не анализирует содержание сообщения. Передаваемые данные понятны только отправителю и получателю и никому более.

Для непосредственной связи друг с другом взаимодействующие процессы используют следующие функции:

Send() // посылка сообщений
Receive() // получение сообщений
Reply() //ответ процессу, пославшему сообщение

Эти функции могут быть использованы как локально, т.е. для связи между процессами на одном компьютере, так и в пределах сети, т.е. для связи между процессами на разных узлах.

Процесс А посылает сообщение процессу В, вызвав функцию Send(), которая передает соответствующий запрос ядру. В этот момент времени процесс А переходит в **SEND-блокированное** состояние и остается в этом состоянии до тех пор, пока процесс В не вызовет функцию Receive() для получения сообщения.



Процесс B вызывает Receive() и получает сообщение от процесса A. При этом состояние процесса A изменяется на **REPLY-блокирован**.

Процесс B при вызове функции Receive() в данном случае не блокируется, т.к. к этому моменту его уже ожидало сообщение от процесса A.

Если бы процесс B вызвал Receive() до того, как ему было послано сообщение, то он бы попал в состояние **RECEIVE-блокирован** до получения сообщения. В этом случае процесс-отправитель сообщения немедленно после посылки сообщения попал бы в состояние REPLY-блокирован.

Процесс B выполняет обработку полученного от процесса A сообщения и затем вызывает функцию Reply(). Ответное сообщение передается процессу A, который переходит в состояние готовности к выполнению.

Вызов Reply() не блокирует процесс B, который также готов к выполнению. Какой из этих процессов будет выполняться, зависит от их приоритетов.

Прокси - это форма неблокирующего сообщения, особенно подходящего для извещения о наступлении события, когда посылающий процесс не нуждается в диалоге с получателем.

Единственная функция прокси состоит в посылке фиксированного сообщения определенному процессу, который является владельцем прокси.

Подобно сообщениям, прокси могут быть использованы в пределах всей сети.

Используя прокси, процесс или обработчик прерывания может послать сообщение другому процессу, не блокируясь и не ожидая ответа.

Прокси может быть использован в следующих случаях:

процесс желает известить другой процесс о наступлении какого-либо события, но при этом не может позволить себе посылку сообщения (в этом случае он оставался бы блокированным до тех пор, пока получатель не вызовет Receive() и Reply());

процесс хочет послать данные другому процессу, но при этом ему не требуется ни ответа, ни какого-либо другого подтверждения того, что адресат (получатель) получил сообщение;

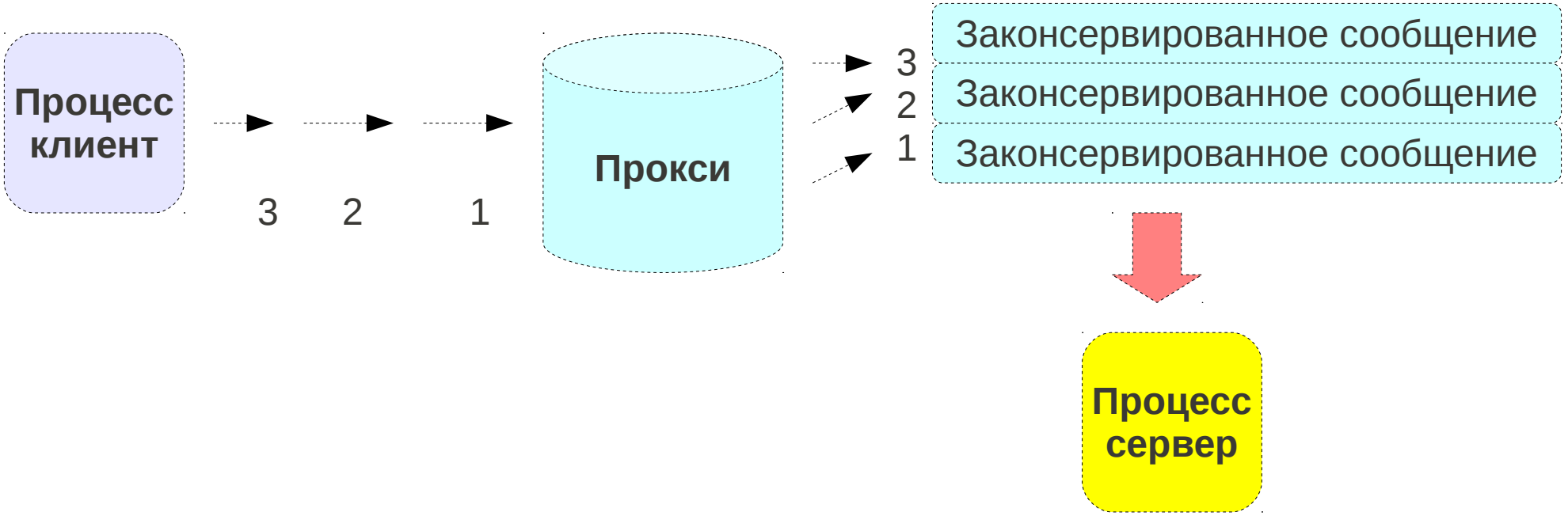
обработчик прерывания хочет известить процесс о поступлении новых данных.

Для создания прокси используется функция **qnx_proxy_attach()**.

Любой другой процесс или обработчик прерывания, которому известен идентификатор прокси, может воспользоваться функцией **Trigger()** для того, чтобы заставить прокси передать заранее заданное сообщение.

Прокси может быть "запущено" неоднократно - каждый раз при этом оно посылает сообщение.

Прокси может накапливать очередь длиной до 65535 сообщений.



Сигналы являются традиционным способом связи, которая используется в течение многих лет в различных операционных системах.

Сигналы во многом схожи с прерываниями. Они позволяют прервать выполнение прикладной программы и отреагировать на породившее сигнал событие.

Сигналы, в частности, используются

при обработке исключений (деление на 0, использование неверного адреса и др.);

для сообщения об асинхронном событии (об окончании операции ввода/вывода, срабатывании таймера и др.);

для организации взаимодействия потоков управления.

Сигналы могут генерироваться (посылаться процессу) как операционной системой, так и процессом.

В системе имеется несколько видов сигналов. Каждому сигналу соответствует уникальное положительное число (номер сигнала). Кроме того, для сигналов определены имена.

Реакция процесса на сигнал:

если процесс не определил никаких специальных мер по обработке сигнала, то *выполняется предусмотренное для сигнала действие по умолчанию* – обычно таким действием по умолчанию является завершение работы процесса;

процесс *может игнорировать* сигнал. Если процесс игнорирует сигнал, то сигнал не оказывает на процесс никакого воздействия;

процесс может предусмотреть *обработчик сигнала* - функцию, которая будет вызываться при приеме сигнала. Если процесс содержит обработчик для какого-либо сигнала, говорят, что процесс может "поймать" этот сигнал. Любой процесс, который "ловит" сигнал, фактически получает особый вид программного прерывания. Никакие данные с сигналом не передаются.

Возможна как асинхронная, так и синхронная обработка сигналов. В случае асинхронной обработки при поступлении сигнала выполнение потока управления приостанавливается и производится обработка сигнала. В этом случае говорят, что сигнал был **доставлен**.

Говорят, что в промежутке времени между генерацией сигнала и его доставкой или приемом он **задержан**.

В промежутке времени между моментом, когда сигнал порожден, и моментом, когда он доставлен, сигнал называется **ожидающим**.

Для процесса ожидающими одновременно могут быть несколько различных сигналов. Сигналы доставляются процессу, когда планировщик ядра делает этот процесс готовым к выполнению.

Асинхронная обработка сигналов может быть временно запрещена. Запрет обработки сигналов производится в рамках потока, а не процесса.

Каждый поток имеет свою собственную **маску сигналов**, обработка которых запрещена (заблокирована,замаскирована).

Если сигнал пришел в тот момент, когда его обработка запрещена, факт прихода сигнала запоминается. В этом случае сигнал будет обработан, когда обработка сигналов будет вновь разрешена, если сигнал не будет ранее обработан синхронным образом.

Пока процесс выполняет обработчик сигнала, этот сигнал автоматически блокируется. Это означает, что нет необходимости заботиться о вложенных вызовах обработчика сигнала. Каждый вызов обработчика сигнала - это неделимая операция по отношению к доставке следующих сигналов этого типа.

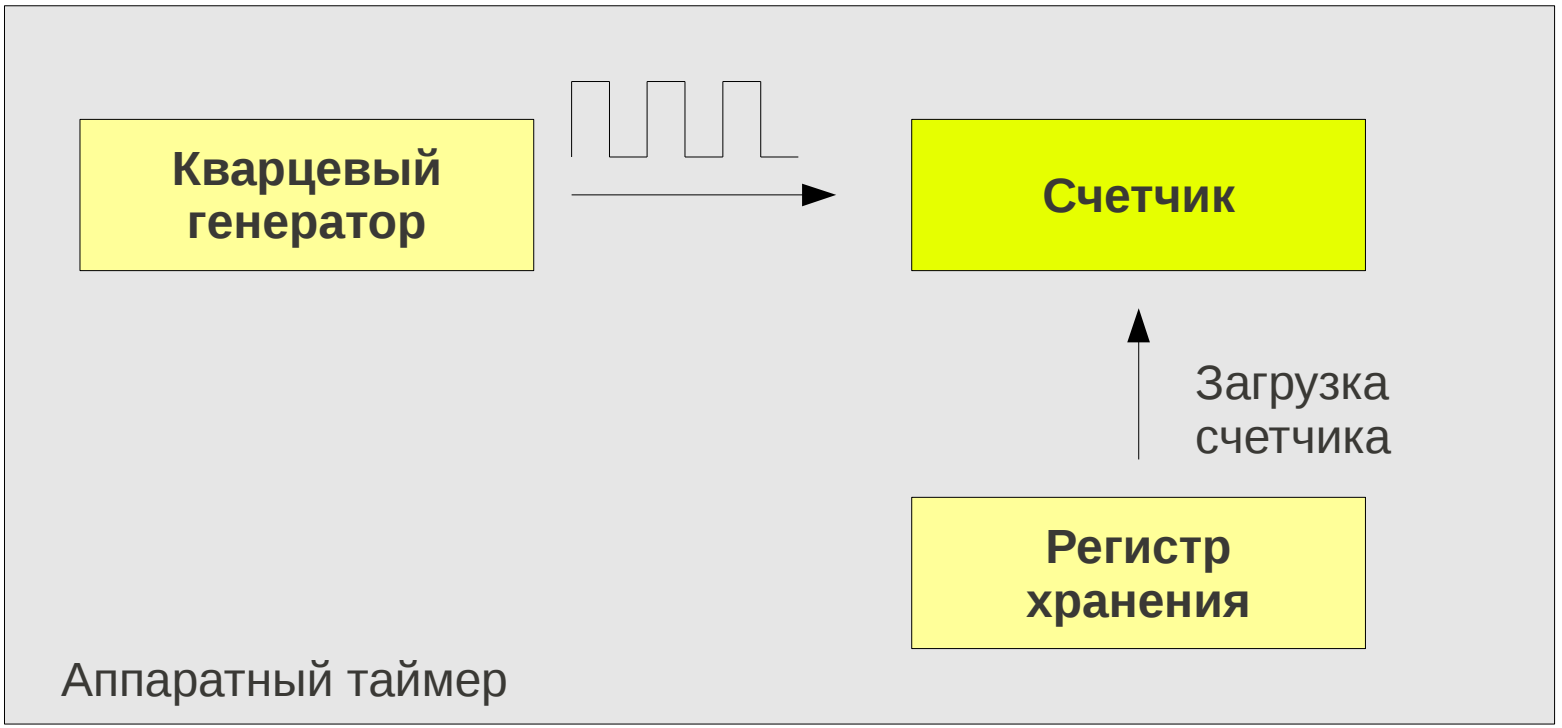
Если процесс выполняет нормальный возврат из обработчика, сигнал автоматически разблокируется.

В случае *синхронной* обработки сигналов поток приостанавливается до тех пор, пока не придет сигнал. Говорят, что сигнал был **принят**, если он был обработан синхронным образом.

Время в ОСРВ

Фиксация временных интервалов (тайм-ауты, тайм-ауты ядра, интервальные таймеры и др.) и хронометраж выполнения участков кода для операционных систем реального времени на порядок более критичны, чем для операционных систем общего назначения.

Кроме того, время весьма важно при запуске задач по расписанию, а также при планировании задач.



У программируемого таймера обычно есть несколько режимов работы.

В режиме **одновибратора** при запуске таймера содержимое регистра хранения копируется в счетчик. Затем содержимое счетчика уменьшается на единицу при каждом импульсе от кристалла. Когда счетчик достигает нуля. он вызывает прерывание и останавливается до тех пор, пока не будет снова явно запущен программным обеспечением.

В режиме **генератора прямоугольных импульсов** при достижении счетчиком нуля также инициируется прерывание, но содержимое регистра хранения автоматически копируется в счетчик, и весь процесс повторяется снова бесконечно.

Преимущество программируемого таймера состоит в том, что частота прерываний от него может управляться программно.

Например, если используется кристалл с частотой колебаний 500 МГц, то счетчик получает импульс каждые 2 нс. При использовании 32-разрядного регистра можно запрограммировать возникновение прерываний через равные интервалы времени от 2 нс до 8.6 с, называемые **тиками**.

Микросхемы программируемых таймеров могут содержать несколько независимых программируемых таймеров.

Все, что делает таймер – это инициирует прерывания через определенные интервалы времени.

Стандарт POSIX определяет, что система всегда содержит, по крайней мере, одни часы с идентификатором **CLOCK_REALTIME** (системные часы).

Значение этих часов интерпретируется как календарное время, то есть время (в секундах и наносекундах), истекшее с 0 часов 1 января 1970 года.

При наличии соответствующего оборудования могут быть созданы дополнительные часы.

Функция **clock_settime()** позволяет установить показания часов, функция **clock_gettime()**- опросить показания часов, а **clock_getres()** - узнать разрешающую способность часов.

Все три функции работают с высокой точностью, так используют структуру timespec, которая позволяет хранить время в секундах и наносекундах.

Стандарт POSIX также определяет, каким образом можно программно создавать и использовать таймеры.

Для создания таймера используется функция **timer_create()**. Одним из аргументов этой функции является структура **sigevent**, которая определяет вид оповещения о срабатывании таймера (например, посылка сигнала или выполнение указанной функции).

Установка и запуск таймера производится функцией **timer_settime()**. Эта функция определяет время первого срабатывания таймера, а также период срабатывания (если требуется периодическое срабатывание таймера).

Модель временной шкалы в операционных системах реального времени основана на следующих предпосылках:

микроядро операционной системы «живет» в дискретной сетке времени;

каждый единичный момент времени для микроядра – это тик системного времени;

какие-либо изменения состояний времени фиксируются микроядром только в узлах этой дискретной шкалы времени;

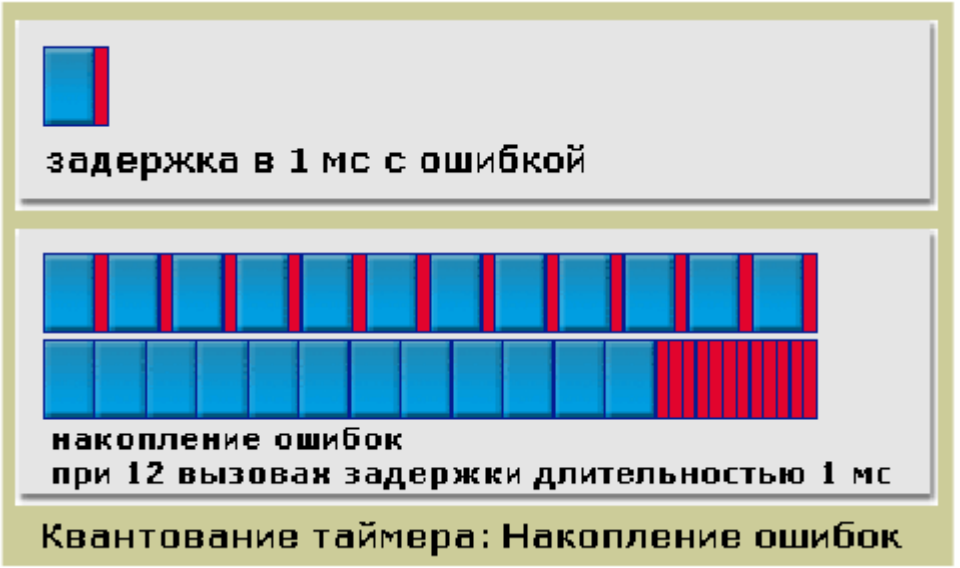
микроядро «не различает» два временных события, если они происходят между соседними тиками системного времени (т.е. с интервалом меньшим, чем интервал системного тика).

Выполнится ли этот код за одну секунду?

```
void OneSecondPause()
{
    for ( i=0; i<1000; i++ )
    delay(1); /* Ждать 1000 милисекунд */
}
```

Этот код не вернет управление через одну секунду на IBM PC. Скорее всего, он выполнится за три секунды. Фактически, когда вы вызываете функции nanosleep() или select() с аргументом в n милисекунд, это может занять от n милисекунд до бесконечности.

Так как мы вызываем delay() асинхронно с прерываением системного таймера, это означает, что мы должны добавить один такт, чтобы убедиться, что мы отмеряем время корректно.



```
void OneSecondPause()
{
    for ( i=0; i<100; i++ ) delay(10);
    // Ждать 1000 милисекунд
}
```

Получится очень близкое к требуемому значение, ошибка будет составлять не более 1/10 секунды.

В ОСРВ множество задач одновременно могут запросить сервис таймера. При отсутствии в системе достаточного количества физических таймеров, они могут смоделированы программно.

Один из способов реализации большого числа виртуальных таймеров состоит в создании таблицы, хранящей все времена сигналов для обрабатываемых таймеров.

При каждом тике обработчик проверяет, не пора ли подавать сигнал от ближайшего таймера. При этом ищется следующий по времени таймер.

Обзор ОСРВ

Свободные:

RTLinux — ОС жёсткого РВ на основе Linux

Android — ОС РВ на основе Linux для мобильных устройств

RTEMS — ОС с открытым исходным кодом, разработана DARPA МО США

OSA — кооперативная многозадачная ОСРВ с открытым исходным кодом для микроконтроллеров PIC (Microchip), AVR (Atmel) и STM8 (STMicroelectronics)

FreeRTOS

Свободные:

KURT (KU Real Time Linux) — ОС мягкого РВ на основе Linux
RTAI

Symbian OS

Xenomai

BeRTOS — ОСРВ для встраиваемых систем с открытым исходным кодом, распространяется по лицензии GPL

Проприетарные:

QNX

ChorusOS

LynxOS

Nucleus

OS-9

RSX-11 (её советский клон — ОСРВ СМ ЭВМ)

VxWorks/Tornado

Windows CE

Virtuoso — ОСРВ для сигнальных процессоров DSP

Багет — ОСРВ разработанная НИИСИ РАН по заказу МО РФ

RTLinux

— **микроядерная операционная система жёсткого реального времени, которая выполняет Linux как полностью вытесняемый процесс.**

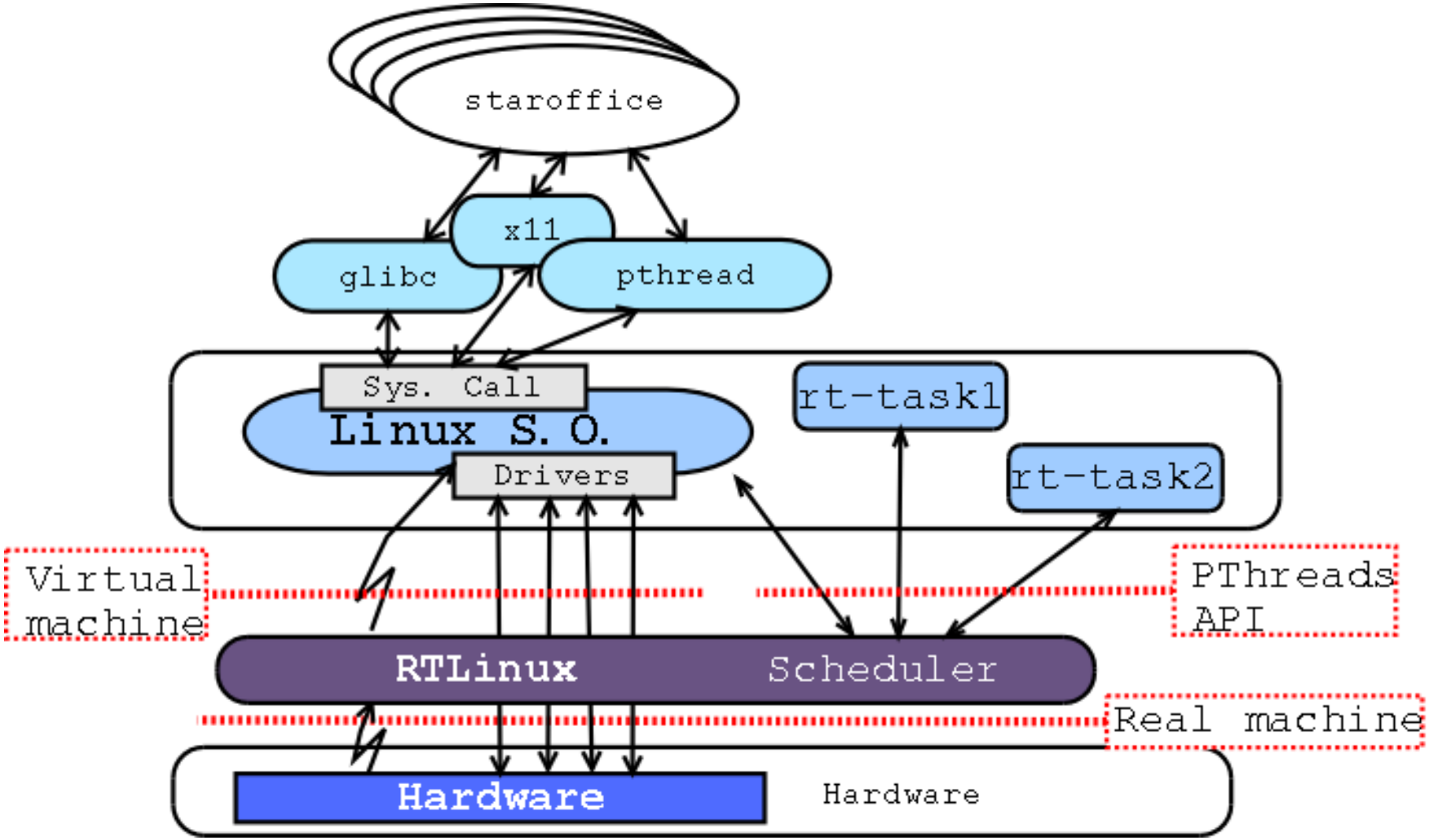
Разработчики RTLinux пошли по тому пути, который предусматривает запуск из **наноядра** реального времени ядра Linux как задачи с наименьшим приоритетом.

В RTLinux все прерывания обрабатываются ядром реального времени, которое включает собственный планировщик задач, обработчик прерываний и библиотечный код.

В случае отсутствия обработчика реального времени для какого-то прерывания, оно передаётся в Linux.

Фактически Linux является простаивающей (idle) задачей ОСРВ, запускаемой только в том случае, если никакая задача не исполняется в реальном времени.

RTLinux - архитектура



Android

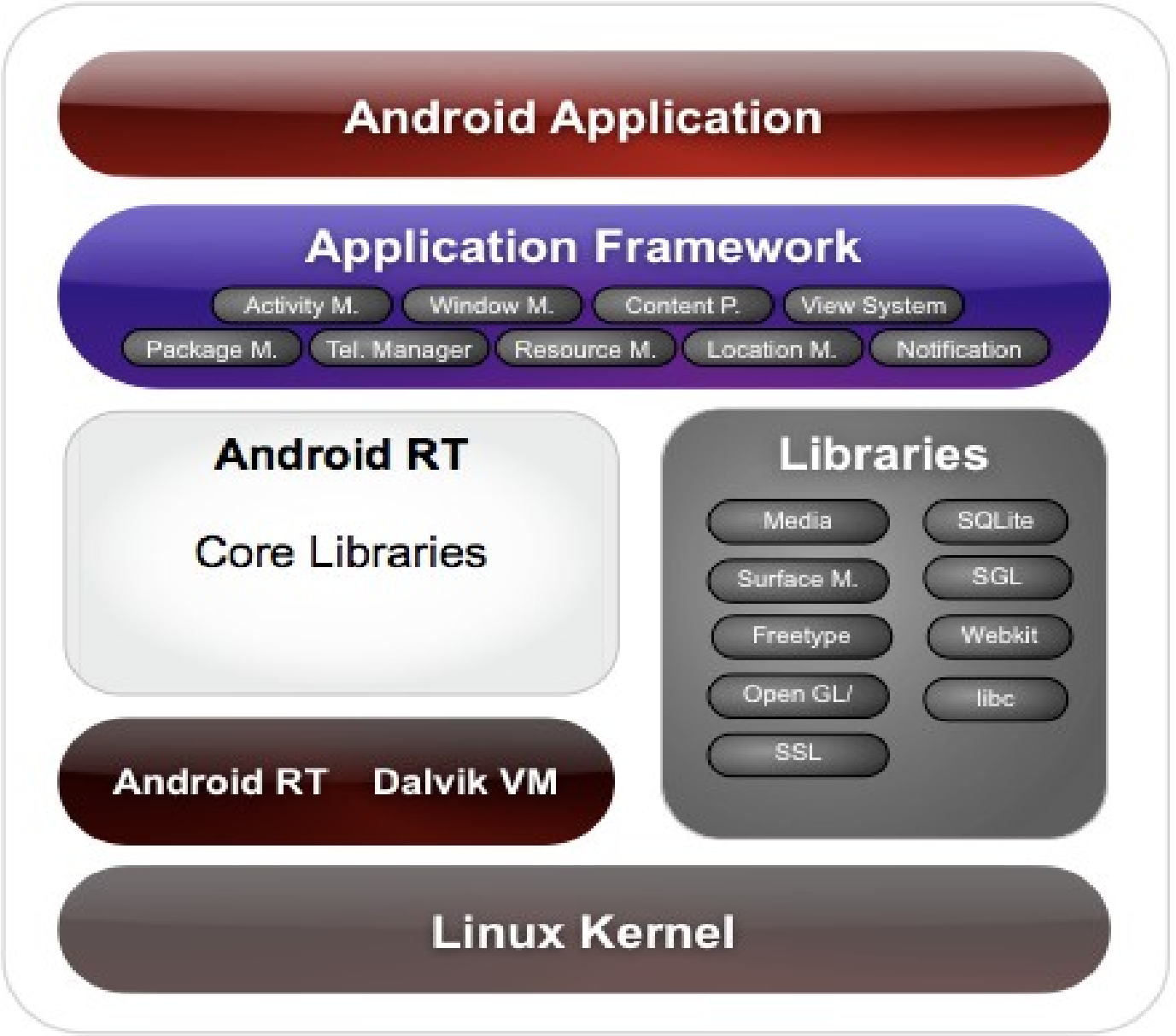
— операционная система для коммуникаторов, планшетных компьютеров, цифровых проигрывателей, цифровых фоторамок, наручных часов, нетбуков и смартбуков, основанная на ядре Linux.

Изначально разрабатывалась компанией Android Inc., которую затем купила Google.

Впоследствии Google инициировала создание альянса Open Handset Alliance (OHA), который сейчас и занимается поддержкой и дальнейшим развитием платформы.

Android позволяет создавать Java-приложения, управляющие устройством через разработанные Google библиотеки. Android Native Development Kit создаёт приложения, написанные на Си и других языках.

Android - архитектура



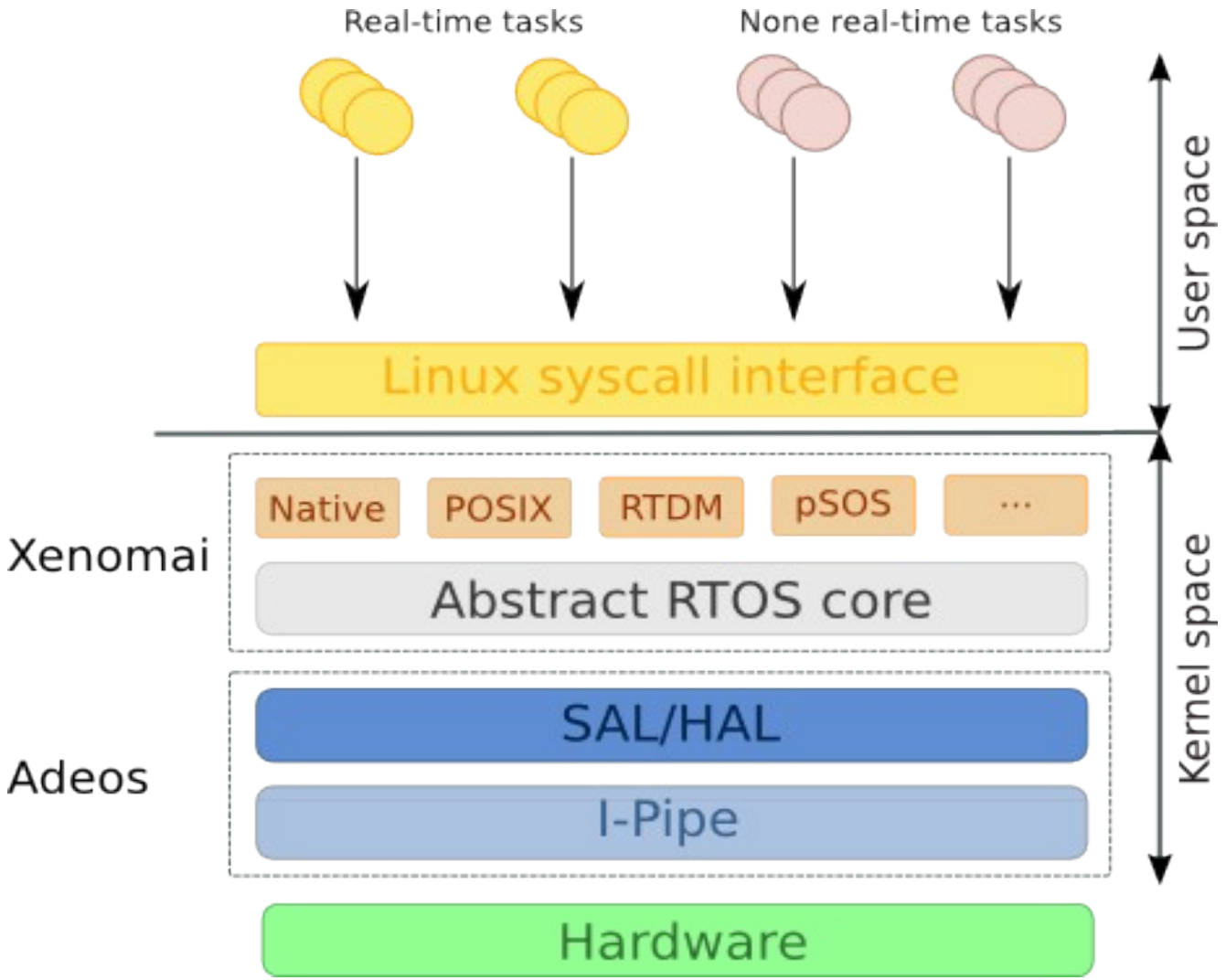
Xenomai

— это фреймворк для разработки приложений реального времени, связанный с ядром Linux, для того, чтобы предоставить всеобъемлющую, с открытым интерфейсом жёсткого реального времени поддержку приложений, легко интегрируемых в окружение Linux.

Проект Xenomai начался в августе 2001 года. В 2003-м он был объединён с проектом RTAI, чтобы предоставить свободную платформу промышленного уровня для Linux, названную RTAI/fusion, на базе ядра Xenomai для абстрактной операционной системы реального времени.

В конечном счёте проект RTAI/fusion стал независимым от RTAI в 2005 году под названием Xenomai.

Xenomai - архитектура



Xenomai - применение

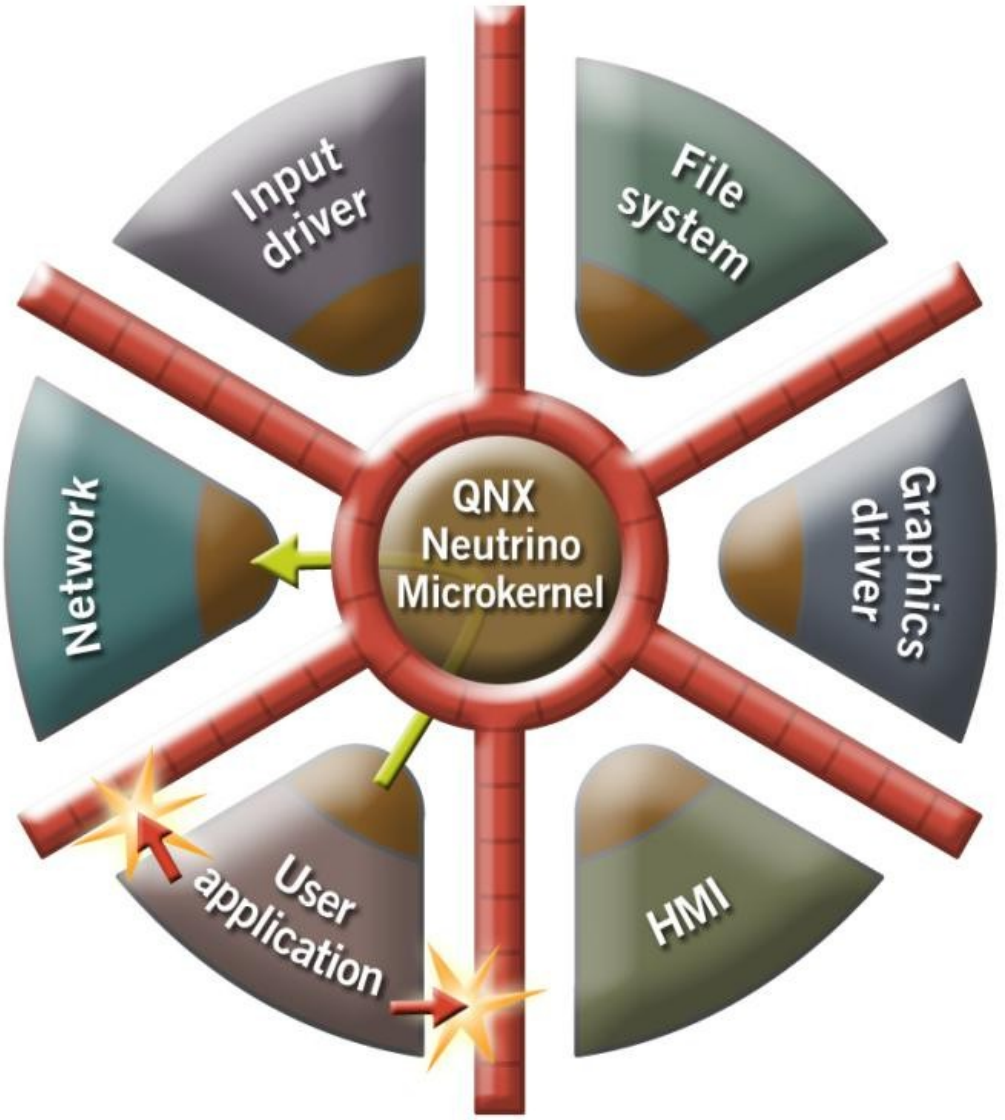
QNX

- POSIX-совместимая операционная система реального времени, предназначенная преимущественно для встраиваемых систем. Считается одной из лучших реализаций концепции микроядерных операционных систем.

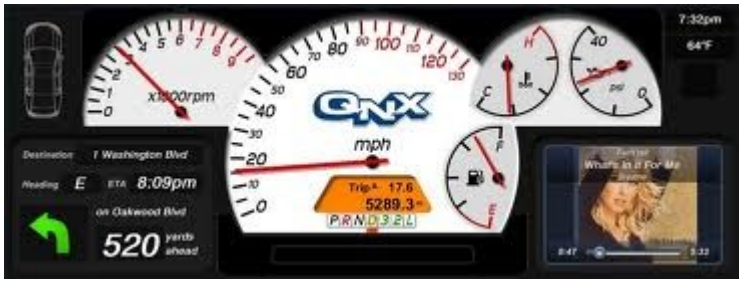
QNX основана на идее работы основной части своих компонентов, как небольших задач, называемых сервисами.

Использование микроядра в QNX позволяет пользователям (разработчикам) отключить любую ненужную им функциональность, не изменяя ядро. Вместо этого можно просто не запускать определённый процесс.

QNX – архитектура микроядра



QNX – применение



Windows CE

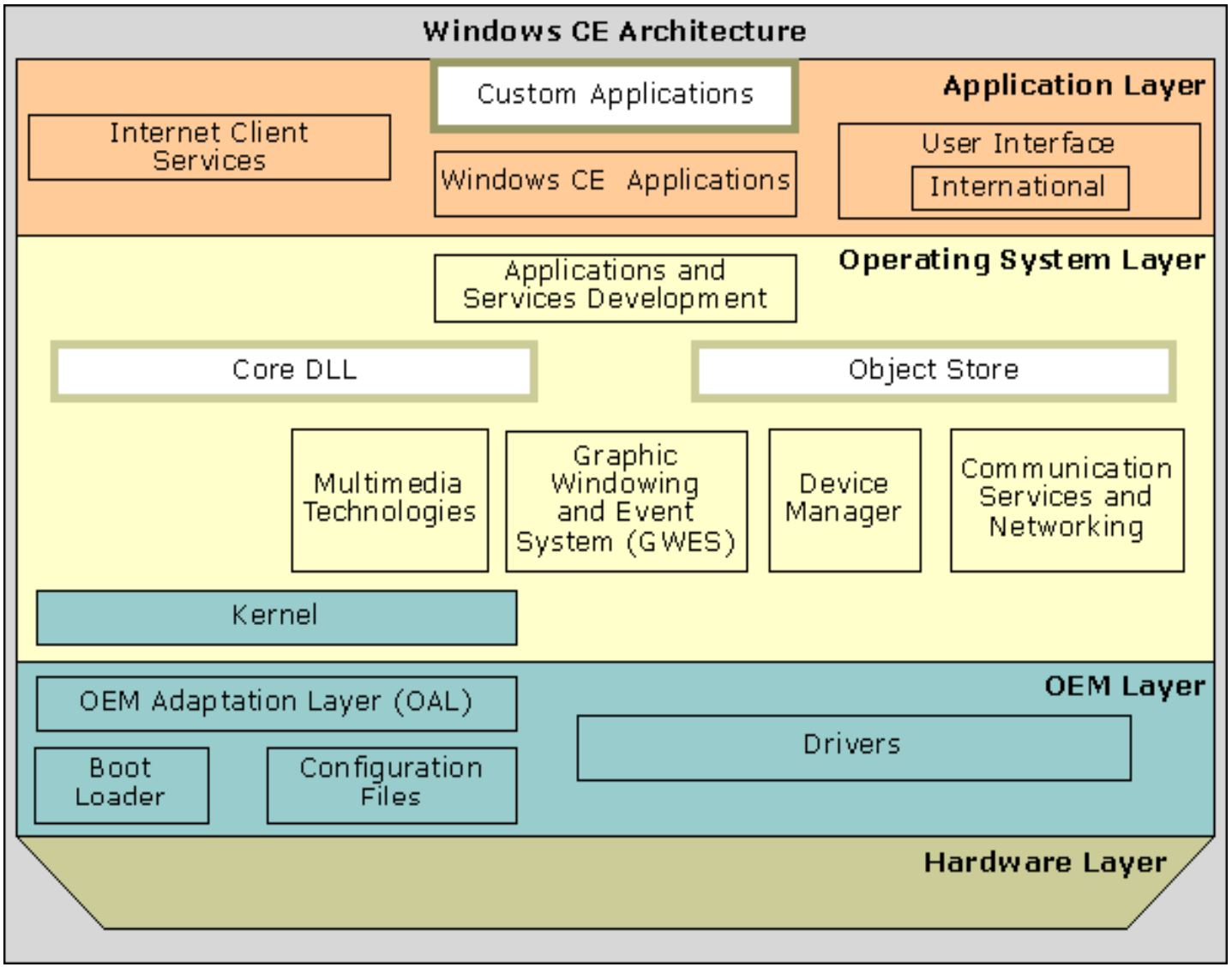
- это вариант операционной системы Microsoft Windows для наладонных компьютеров, мобильных телефонов и встраиваемых систем.

Сегодня **Windows CE** (**Consumer Electronics** — бытовая техника) не является «урезанной» версией Windows для настольных ПК, она основана на совершенно другом ядре и является операционной системой реального времени с набором приложений, основанных на Microsoft Win32 API.

Windows CE — это компонентная, многозадачная, многопоточная, многоплатформенная операционная система **с поддержкой реального времени.** Разработчикам доступны около 600 компонентов, используя которые они могут создавать собственные образы операционной системы, которые включает только необходимый данному конкретному устройству функционал.

Windows CE оптимизирована для устройств, имеющих минимальный объём памяти: ядро Windows CE может работать на 32 КБ памяти. С графическим интерфейсом (GWES) для работы Windows CE понадобится от 5 МБ. Устройства часто не имеют дисковой памяти и [illegible] сконструированы как «закрытые» устройства, без [illegible] расширения пользователем (например, ОС может быть «зашит

Windows CE – архитектура

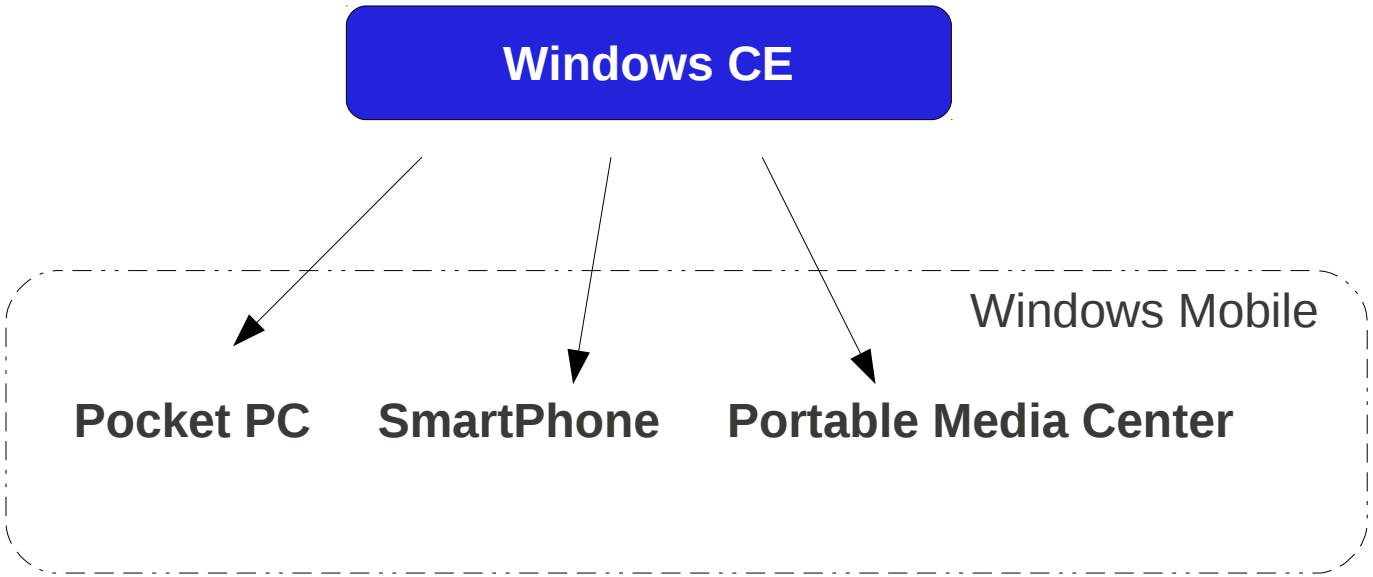


Часто названия **Windows CE, Windows Mobile, Pocket PC** используют как взаимозаменяемые. Это не совсем правильно.

Windows CE — это модульная операционная система, которая служит основой для устройств нескольких классов.

Любой разработчик может купить инструментарий (Platform Builder), который содержит все эти компоненты и программы, позволяющие построить собственную платформу. При этом такие приложения, как Word Mobile / Pocket Word, не являются частью этого инструментария.

Windows Mobile лучше всего представлять себе как набор платформ, основанных на Windows CE. В настоящее время в этот набор входят платформы: **Pocket PC, SmartPhone и Portable Media Center**. Каждая платформа использует свой набор компонентов Windows CE, плюс свой набор сопутствующих особенностей и приложений.



Windows CE – применение

VxWorks

- операционная система реального времени, разрабатываемая компанией Wind River Systems (США) (приобретена компанией Intel 17 июля 2009 г.), ориентированная на использование в встраиваемых компьютерах, работающих в системах жёсткого реального времени.

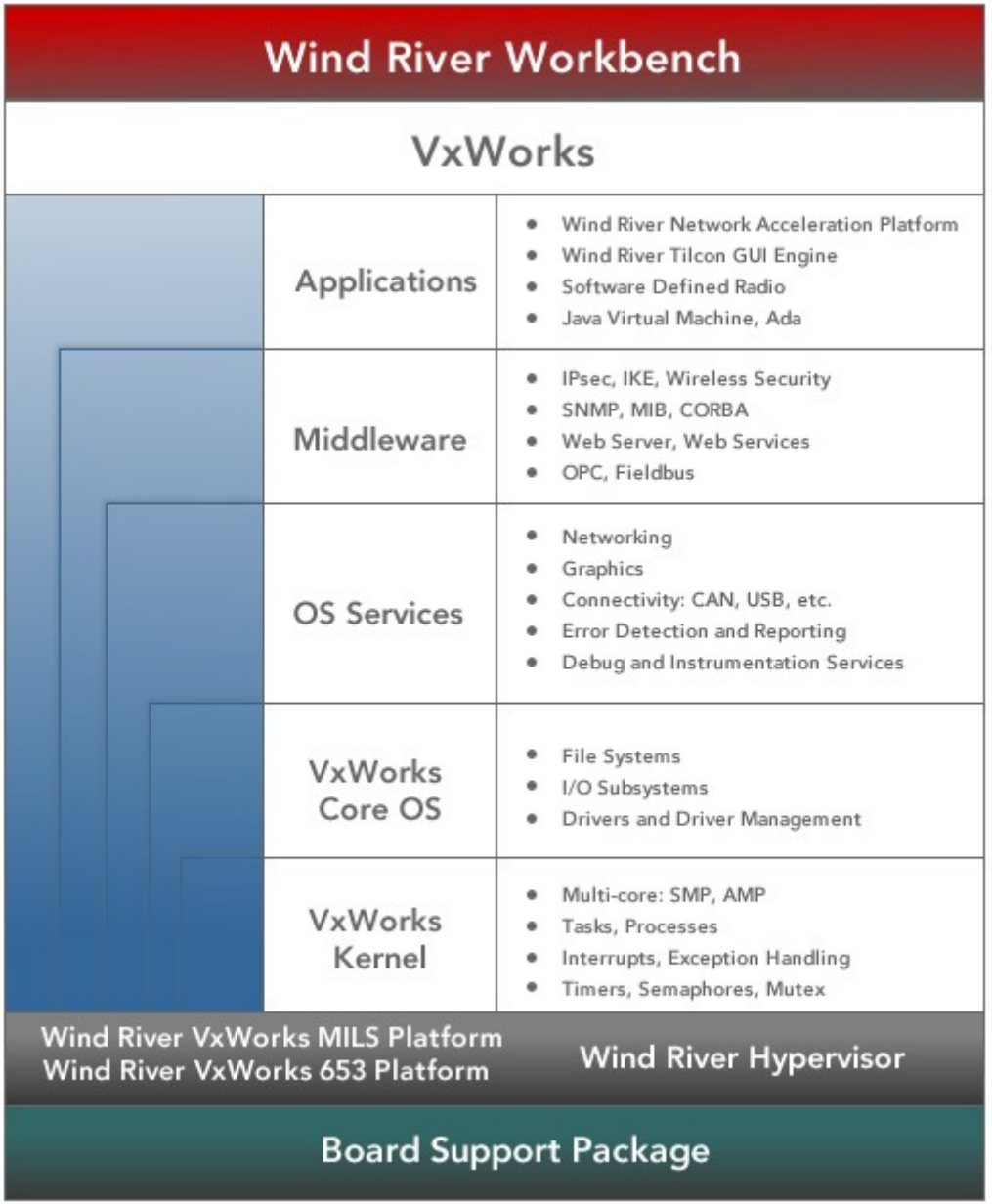
VxWorks является системой с кросс-средствами разработки прикладного программного обеспечения. Иначе говоря, разработка происходит на инструментальном компьютере, называемом **host**, для последующего применения его на целевой машине — **target**.

VxWorks имеет архитектуру клиент-сервер и, как и большинство ОС жёсткого реального времени, построена по технологии микроядра.

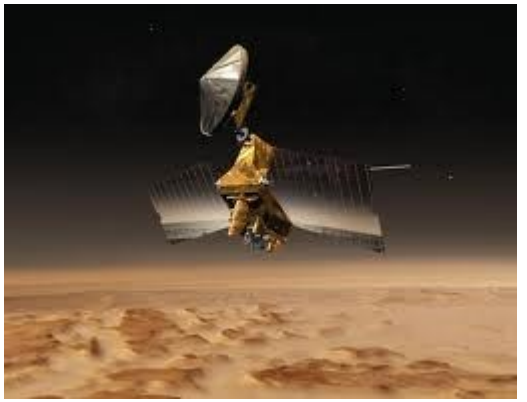
На самом нижнем непрерываемом уровне ядра (WIND Microkernel) выполняются только базовые функции планирования задач и управления коммуникацией/синхронизацией между задачами. Все остальные функции ОСРВ более высокого уровня — управление памятью, сетевые средства и т. д. — реализуются через простые функции нижнего уровня. За счёт такой иерархической организации достигается быстродействие и детерминированность ядра системы, также это позволяет легко строить необходимую конфигурацию операционной системы.

VxWorks®
WIND RIVER

VxWorks – архитектура



VxWorks – применение



SCADA

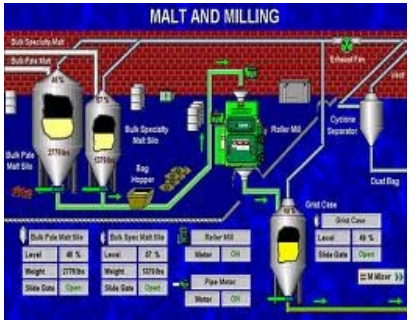
Supervisory Control And Data Acquisition, Диспетчерское управление и сбор данных — программный пакет предназначенный для разработки или обеспечения работы в реальном времени систем сбора, обработки, отображения и архивирования информации об объекте мониторинга или управления.

SCADA-системы используются там, где требуется обеспечивать операторский контроль за технологическими процессами **в реальном времени**.

Термин SCADA имеет двоякое толкование. Наиболее широко распространено понимание SCADA как приложения, то есть программного комплекса, обеспечивающего выполнение указанных функций, а также инструментальных средств для разработки этого программного обеспечения.

Однако, часто под SCADA-системой подразумевают программно-аппаратный комплекс.

В 80-е годы под SCADA-системами чаще понимали программно-аппаратные комплексы сбора данных реального времени. С 90-х годов термин SCADA больше используется для обозначения только программной части *человеко-машинного интерфейса* (HMI).



SCADA - задачи

Обмен данными с промышленными контроллерами и платами ввода/вывода в реальном времени через драйверы.

Обработка информации **в реальном времени**.

Логическое управление.

Отображение информации на экране монитора **в удобной и понятной для человека** форме.

Ведение базы данных реального времени с технологической информацией.

Аварийная сигнализация и управление тревожными сообщениями.

Подготовка и генерирование **отчетов** о ходе технологического процесса.

Осуществление сетевого **взаимодействия** между SCADA ПК.

Обеспечение связи с **внешними приложениями**.

SCADA - компоненты

Драйверы или серверы ввода-вывода

Система реального времени

Человеко-машинный интерфейс (HMI - Human Machine Interface)

Программа-редактор для разработки HMI.

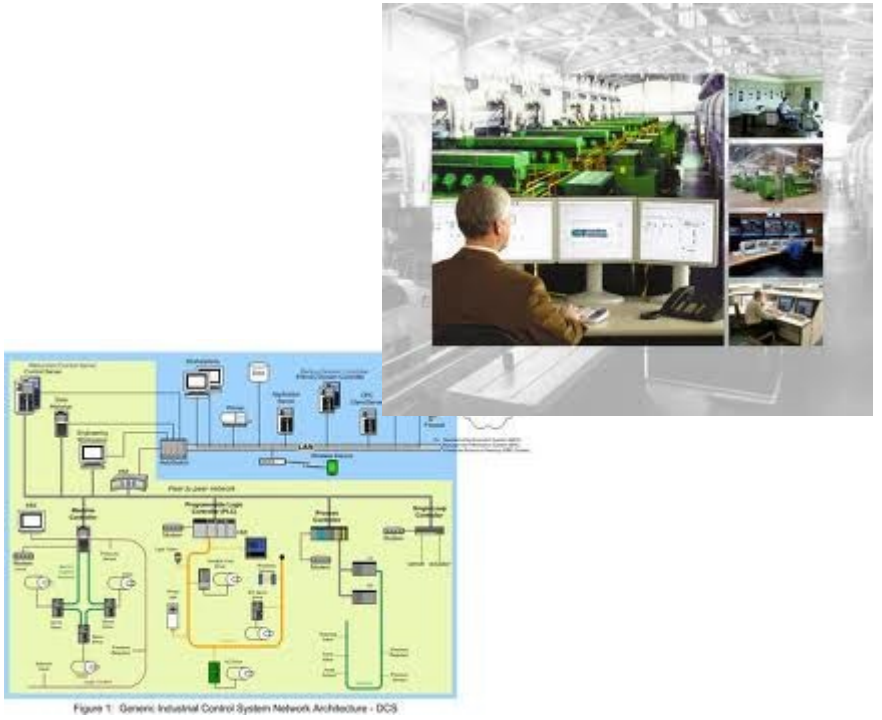
Система логического управления

База данных реального времени

Система управления тревогами

Генератор отчетов

Внешние интерфейсы



SCADA - PLC

Программируемый логический контроллер (ПЛК) или Programmable Logic Controller (PLC) или программируемый контроллер — электронная составляющая промышленного контроллера, используемого для автоматизации технологических процессов.

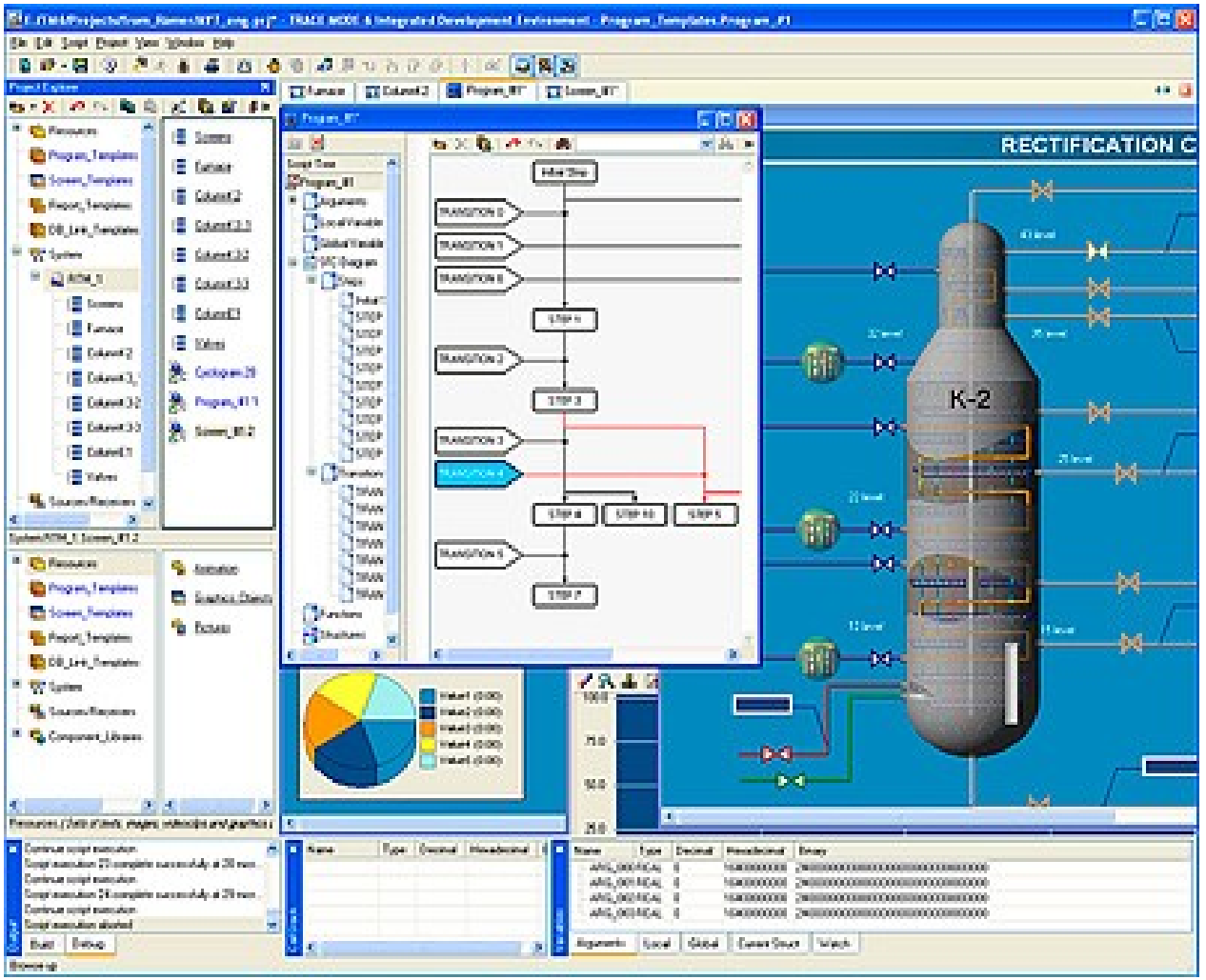
Условия работы ПЛК:

неблагоприятная окружающая среда,

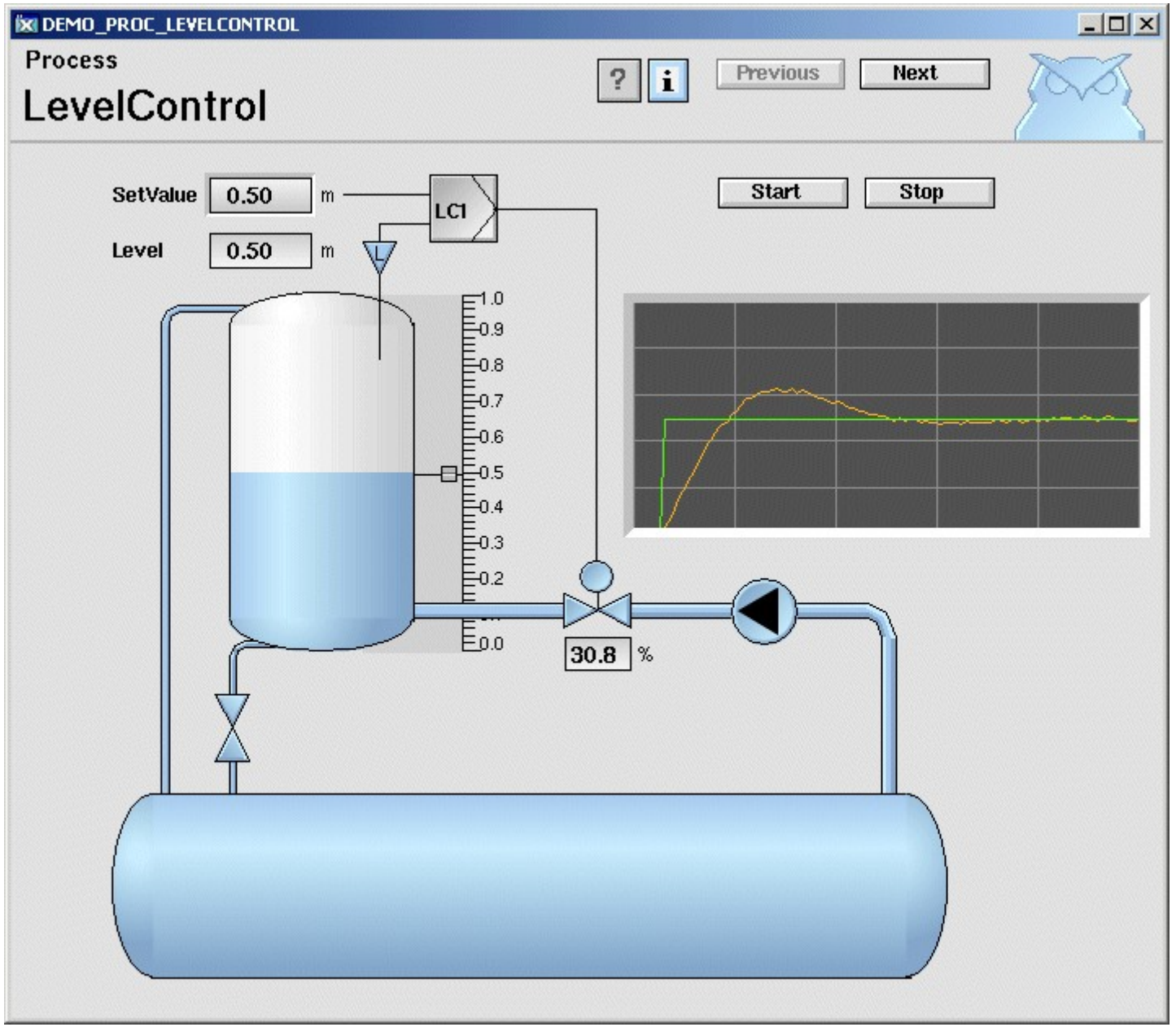
автономное использование, без серьёзного обслуживания и практически без вмешательства человека.

ПЛК являются устройствами **реального времени**

TRACE MODE



Proview


DEMO_PROC_LEVELCONTROL
Process
LevelControl
Previous
Next
SetValue
0.50
m
Level
0.50
m
LC1
Start
Stop
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
30.8
%